Universally Composable Key-Management

Steve Kremer[1], Robert Künnemann[2], and Graham Steel[2]

[1] LORIA & INRIA Nancy – Grand-Est
[2] LSV & INRIA Saclay – Île-de-France

Abstract. We present the first key-management functionality in the Universal Composability (UC) framework. It allows the enforcement of a wide range of security policies and can be extended by diverse key usage operations with no need to repeat the security proof. We illustrate its use by proving an implementation of a Security API secure with respect to arbitrary key-usage operations and explore a proof technique that allows the storage of cryptographic keys externally, a novel development in the UC framework.

Keywords: Key management, Security APIs, Universal Composability

Table of Contents

1 Introduction

Security critical applications often store keys on dedicated hardware security modules (HSM) or key-management servers to separate highly sensitive cryptographic operations from more vulnerable parts of the network. Access to such devices is given to protocol parties by the means of *Security APIs*. Examples of such APIs are the RSA PKCS#11 standard [RSA04], IBM's CCA [CCA06] and the trusted platform module (TPM) [TCG07] API. Building on the work of Longley and Rigby [LR92] and Bond and Anderson [BA01] on API attacks, several recent papers have investigated the security of APIs on the logical level, often using symbolic techniques for protocol analysis, e. g., [BCFS10,CKS07,DKS10]. More recent work has tried to define appropriate security notions for APIs in terms of cryptographic games [CC09,KSW11]. This has two major disadvantages: first, it is not clear how the security notion will compose with other protocols implemented by the API, and second, it is difficult to see whether a definition covers the attack model completely, since the game can be tailored to a design. For example, the security game may not permit re-importing wrapped keys [CC09]. This illustrates that game-based definitions are very difficult to get right in this domain. Since security APIs are first and foremost used as building blocks in other protocols, composability is crucial. In this work, we will follow the more general approach to API security of Kremer et al. [KSW11], but using a framework that allows for composition.

Composability can be proven in frameworks for simulation-based security, such as Universal Composability (UC) [Can05]. The requirements of a protocol are

formalized by abstraction: an *ideal functionality* computes the protocol's inputs and outputs securely, a 'secure' protocol is one that emulates the functionality. Simulation-based security naturally models the composition of the API with other protocols, so that proofs of security can be performed in a modular fashion.

Over time a number of shortcomings have been found in the UC framework, leading to the contribution of other frameworks in the same spirit [BDHK07,HS11,Küs06,MR11]. Some particular weaknesses of the UC framework are the following: First, it is assumed that sessions of running protocols are identifiable by a (pre-established) session identifier, abbreviated as *sid*, although it is possible to define composition on protocols without such identifiers. For a Security API, one could argue that a serial number constitutes such an identifier. Still, the identifier is part of every incoming and outgoing message, which is a rather unrealistic modelling of locality information. Second, adaptive corruption of parties, or of keys that produce an encryption, provokes the well-known commitment problem [Hof08], so we will have to place some limitations on the types of corruptions that can occur. We therefore restrict the environment to avoid these problems. Third, the proof of the composition theorem is flawed due to an inadequate formulation of the composition operation [HS11], though here the authors remark that, "none of the objections we raise point to gaps in security proofs of existing protocols. Rather, they seem artifacts of the concrete technical formulation of the underlying framework.". We employ UC despite its shortcomings, because it is well-studied and hence provides a common ground for transferring this first approach on modelling key-management to other frameworks.

Contributions. We present, to the best of our knowledge, the first composable definition of secure key-management in form of a key-management functionality $\mathcal{F}^{\mathrm{KM}}$. It assures that keys are transferred correctly from one Security API to another, that the policy is respected and that operations which use keys are computed correctly. The latter is achieved by describing operations unrelated to key-management by so-called key-usage functionalities. $\mathcal{F}^{\mathrm{KM}}$ is parametric in the policy and the set of key-usage functionalities, which can be arbitrary. This facilitates revision of security devices, because changes to operations that are not part of the key-management or the addition of new functions does not affect the UC emulation proof. In order to achieve this extensibility, we investigate what exactly a "key" means in simulation-based security. Common functionalities in UC do not allow two parties to share the same key, in fact, they do not have a concept of keys, but a concept of 'the owner of a functionality' instead. The actual key is kept in the internal state of a functionality, used for computation, but never output. Dealing with key-management, we need the capability to export and import keys and we propose an abstraction of the concept of keys, that we call *credentials*. The owner of a credential can not only compute a cryptographic operation, but he can also delegate this capacity by transmitting the credential. We think this concept is of independent interest, and as a further contribution, subsequently introduce a general proof method that allows the substitution of credentials by actual keys when instantiating a functionality.

Some aspects of the ideal functionality $\mathcal{F}_{\text{crypto}}$ by Küsters et al. [KT11] are similar to our key-management functionality in that both provide cryptographic primitives to a number of users and enjoy composability. However, the $\mathcal{F}_{\text{crypto}}$ approach aims at abstracting a specified set of cryptographic operations on client machines to make the analysis of protocols in the UC model easier, and does not address key-management nor policies.

Limitations. The key-management functionality is tightly coupled with a functionality $\mathcal{F}^{\text{KW}}$ used to transfer keys from one device to another. This functionality describes a deterministic authenticated symmetric encryption scheme that is secure against key-dependant messages. While deterministic, symmetric authenticated encryption is indeed typically used to transfer keys (see, e. g., RFC 3394), it restricts the analysis to security devices providing this kind of encryption. We have not yet covered asymmetric encryption of keys in $\mathcal{F}^{\text{KW}}$ (as opposed to asymmetric encryption of user-supplied data), although $\mathcal{F}^{\text{KW}}$ could in theory be altered to support this.

2 A key-management functionality

In this section we introduce our key-management functionality $\mathcal{F}^{\text{KM}}$. We will work in the 2005 version of *Universal Composability* (UC) framework [Can05], described briefly below. We do not want to hard code the cryptographic operations, which are orthogonal to the key management part that can be provided by our security API. Therefore we introduce the notion of *key-usage functionalities*. Finally, we will discuss the formalization of *security policies*, which define the expected security guarantees. At the end of this section we give a detailed description of the key-management functionality $\mathcal{F}^{\text{KM}}$.

2.1 Universal Composability

In the UC framework, the security goal of a protocol is specified by a so-called *ideal functionality*, which acts as a third party and is trusted by all participants. A set of users, the protocol parties, as well as a distinguished user, the *adversary*, interact with the functionality using private, authenticated channels. Using the inputs of the parties, the ideal functionality defines a secure way of computing anything the protocol shall compute, explicitly computing the data that is allowed to leak to the attacker. For instance, a secret channel is specified as a functionality that takes a message from Alice and sends it to Bob, notifying the attacker of the length of the message, which would be leaked if this channel were to be realized using encryption. Each of the parties and functionalities are interactive probabilistic polynomial time (PPT)—in a given complexity parameter η—Turing machines, communicating via tapes. A protocol is a network of interactive PPT Turing Machines that communicate with each other, but receive protocol inputs and send protocol outputs to the environment. Roughly, security of a protocol is expressed as indistinguishability of the protocol from an ideal system. We introduce another

party, the environment $\mathcal{E}$, whose role is to distinguish whether it is interacting with the ideal system (users interacting with an ideal functionality) or the real system (users executing a protocol). We say that a protocol π *UC-emulates* a functionality $\mathcal{F}$ if for all attackers interacting with π, there exists an attacker, the simulator *Sim*, interacting with $\mathcal{F}$, such that no environment can distinguish between interacting with the attacker and the real protocol π, or the simulation of this attack (generated by *Sim*) and $\mathcal{F}$. Canetti [Can05] showed that it is actually not necessary to quantify over all possible adversaries: the most powerful adversary is the attacker that merely acts as a relay forwarding all messages between the environment and the protocol. This attacker is called the dummy-attacker $\mathcal{D}$. We will denote by $\mathrm{EXEC}^{\eta}_{\pi,\mathcal{A},\mathcal{E}}$ (respectively $\mathrm{IDEAL}^{\eta}_{\mathcal{F},\mathcal{A},\mathcal{E}}$) the distribution over traces generated by the execution of the system consisting of the protocol π (respectively a set of users interacting with the ideal functionality $\mathcal{F}$), the adversary $\mathcal{A}$ and the environment $\mathcal{E}$. $(\mathrm{EXEC}^{\eta}_{\pi,\mathcal{A},\mathcal{E}})_{\eta}$ and $(\mathrm{IDEAL}^{\eta}_{\mathcal{F},\mathcal{A},\mathcal{E}})_{\eta}$ denote the families of distributions (ensembles) indexed by the complexity parameter η. We denote by $(D_1)_{\eta} \approx (D_2)_{\eta}$ the usual notion of indistinguishability of ensembles. This allows us to define UC realizability.

Definition 1 (UC realization). *Let π and $\mathcal{F}$ be PPT Turing machines.* π UC-emulates $\mathcal{F}$*, denoted* $\pi \leq_{UC} \mathcal{F}$*, if* $\exists S.\, \forall \mathcal{E} : (\mathrm{EXEC}^{\eta}_{\pi,\mathcal{D},\mathcal{E}})_{\eta} \approx (\mathrm{IDEAL}^{\eta}_{\mathcal{F},Sim,\mathcal{E}})_{\eta}$

Canetti [Can05] has also shown that this notion of security is compositional and that ideal functionalities can be replaced by their realization in an arbitrary context, while preserving security properties. In some cases, in particular to handle some forms of key corruption, we will have to assume conditions on the environment. Then, of course, composition is limited to environments where those conditions can be guaranteed to hold.

Definition 2 (Conditional UC realization). *Let π and $\mathcal{F}$ be PPT Turing machines.* π UC-emulates $\mathcal{F}$ *under condition C, denoted* $\pi \leq^{C}_{UC}, \mathcal{F})$ *if* $\exists S.\, \forall \mathcal{E}$:

$$\left|\Pr[\mathrm{EXEC}^{\eta}_{\pi,\mathcal{D},\mathcal{E}} = 1|C] - \Pr[\mathrm{IDEAL}^{\eta}_{\mathcal{F},Sim,\mathcal{E}} = 1|C]\right|$$

is negligible in η and C is the event that the view of $\mathcal{E}$ fulfils some predicate.

2.2 Global Structure

One of the main problems in key-management is that a distributed system of security tokens $ST_1 \ldots ST_m$ has to implement a security policy *globally*: $ST_1 \ldots ST_m$ should behave like a global key-storage device that everyone can access securely and that assures that the policy is not violated. We use the concept of UC emulation to express exactly that. We define a key-management functionality $\mathcal{F}^{\mathrm{KM}}$, detailed below in Listing 1.1, which is an abstraction of our expectation of key-management.

In Figure 1, we sketch how $ST_1,\ldots,ST_m$ UC-emulate the functionality in a network with an environment $\mathcal{E}$, a dummy attacker $\mathcal{D}$, respectively a simulator $\mathcal{S}$

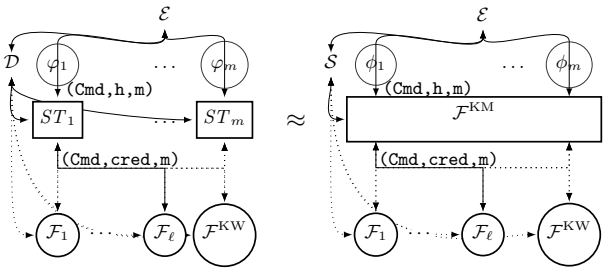

Fig. 1. Distributed security tokens in the network (left-hand side) and idealized functionality $\mathcal{F}^{\mathrm{KM}}$ in the same network (right-hand side), simplified.

and a number of other functionalities (depicted as circles) that will be presented below. The environment accesses the security tokens, respectively $\mathcal{F}^{\mathrm{KM}}$ through *dummy users* φ_i and ϕ_i: they take inputs from the environment and send them to the Security API they are connected to (and vice versa). The only difference between φ_i and ϕ_i is that φ_i forwards its input to ST_i, while ϕ_i forwards it to $\mathcal{F}^{\mathrm{KM}}$. These users are needed to hide that in the ideal system the environment interacts with a single machine while in the distributed implementation it interacts with different security tokens.We will denote the network identity of both φ_i and ϕ_i, with U_i and the adversary's network identity with $\mathcal{A}$.

To transfer a key from one security token to another in the real world, the environment instructs, for instance, U_1 to ask for a key to be *wrapped*. A wrapping of a key is the encryption of a key with another key, the wrapping key, so that it can be transferred from one security token to another without revealing the key. The wrapping key must of course be on both security tokens prior to that. U_1 will receive the wrap from ST_1 and forward it to the environment, which in turn instructs U_2 to unwrap the data it just received from U_1. In the ideal world, $\mathcal{F}^{\mathrm{KM}}$ enforces that, if the wrapping that was just sent to U_1 is unwrapped by U_2, then U_2 gains access to the key that was only available to U_1, and that attributes associated with that key stay the same. ST_1 and ST_2 have to assure the same properties, as otherwise the environment could distinguish between the real world and the ideal world.

2.3 Key-Usage Functionalities

We classify cryptographic operations by whether they operate on user-data (with keys), or on keys (with other keys). We call the latter *key-management*, the former *key-usage*. We are specific about how keys are managed: we require the key-management task to be solved using deterministic, authenticated, symmetric encryption, provided by the functionality $\mathcal{F}^{\mathrm{KW}}$ (detailed in Appendix A). However, for key-usage operations we do not care for much more than their correct implementation, thus keeping $\mathcal{F}^{\mathrm{KM}}$ extensible by all kinds of cryptographic primitives.

In UC, a cryptographic operation is specified by a functionality $\mathcal{F}$, a fact that we will exploit for the definition of $\mathcal{F}^{\mathrm{KM}}$. Let $\mathcal{F}_1,\ldots,\mathcal{F}_l$ be the key usage functionalities (also depicted in Figure 1). For every key-usage operation, $\mathcal{F}^{\mathrm{KM}}$ calls the corresponding KU-functionality, receives the response and outputs it to the user. We define $\mathcal{F}^{\mathrm{KM}}$ for arbitrary key-usage operations, and consider a security token secure, with respect to a certain policy and the implemented KU-functionalities, if it emulates the ideal functionality $\mathcal{F}^{\mathrm{KM}}$ parametrized by those KU-functionalities. This allows us to provide an implementation for secure key-management with respect to a certain policy, and arbitrary key-usage operations, see Section 3. We achieve composability in the following sense: if a device performs the key-management according to our implementation, it does not matter how many, and which functionalities it enables access to, as long as those functionalities provide the amount of security the designer aims to achieve (cf. Theorem 12). Notably, the KU-functionalities appear on both sides in Figure 1. In Section 4, we show how to make use of the UC Theorem to instantiate a KU-functionality by an implementation that UC-emulates it.

This approach imposes some mild assumptions on the KU-functionalities. The computation of a KU-functionality $\mathcal{F}$ must be independent of the value of the caller's identity, as otherwise a response forwarded by $\mathcal{F}^{\mathrm{KM}}$ might differ from a response forwarded by ST_i. Many of the functionalities in, for example, [Can05] bind the roles of the parties, e. g., signer and verifier, to network identities . An implementation of those functionalities usually employs keys. However such functionalities are not caller-independant. They fail to capture the fact that a key allows to pass the *capacity* to, e. g., generate valid signatures for a certain verification key from one ST to another.Therefore, the privilege to perform an operation must be transferable as some piece of information, which cannot be the actual key, if we want to prove such a functionality to be secure: a signing functionality, for example, that exposes its keys to the environment is not realizable, since the environment could then generate dishonest signatures itself. The solution is to generate a key, but only send out a *credential*, which is a hard-to-guess pointer that refers to this key. Whoever knows the credential is allowed to sign. Keys not only allow a distinguished party, the owner of the key, to perform operation but also allow *delegation* of this capacity. Abstraction of 'keys' by 'credentials' is thus more powerful than abstraction of the 'owner of a key' by a 'fixed identity of party that is allowed to sign'. In our scenario, where keys are exported, it is necessary to abstract keys. The requirements on a KU-functionality are formalized by the following definitions.

Definition 3 (caller-independence). *A functionality is* $\mathcal{F}$ caller-independent *iff at any point of the execution of* $\mathcal{F}$ *with any environment, if* m *is a message and* id*,* id' *are two identities of network parties, different from the adversary, then the distributions of the outputs of* $\mathcal{F}$ *after receiving* m *from* id*, respectively* id'*, are the same.*

Definition 4 (key-manageable functionality). *A functionality* $\mathcal{F}$ *is* key-manageable *iff it accepts requests of the form* (Command, sid, m) *for a finite set of*

commands $\mathcal{C}$ *containing at least* Generate, Corrupt $\in \mathcal{C}$ *and* CorruptionStatus $\in \mathcal{C}$*, and if it is* caller independent.

The key-management functionality $\mathcal{F}^{\mathrm{KM}}$ expects the response to Generate to be of the form $(\text{Generate}^{\bullet}, sid, \text{credential})$.

Remark 1: Since keys might be used to wrap other keys, we would like to know how the loss of a key to the adversary affects the security of other keys. When an adversary "corrupts a key" in $\mathcal{F}^{\mathrm{KM}}$, he learns the credentials to access the functionalities. The functionalities will not be notified immediately. However, the adversary is able to use the credentials to corrupt the key in the functionality.

Remark 2: We require that the credentials for different KU-functionalities are distinct. It is nonetheless possible to encrypt and decrypt arbitrary credentials using $\mathcal{F}^{\mathrm{KW}}$. Suppose a designer wants to prove a Secure API that uses shared keys for different operations secure. One way or another, she would need to prove that those roles do not interfere. For this case, we suggest providing a functionality that combines the two KU-functionalities, and proving that the implementation of the two operations combined UC-emulates the combined functionality. It is possible to assign different attributes to keys of the same KU-functionality, and thus restrict their use to certain commands, effectively providing different roles for credentials to the same KU-functionality. This can be done by specifying two attributes for the two roles and defining a policy that restricts which operation is permitted for a key of each attribute.

Remark 3: Many commonly used UC-functionalities are not *caller-independent*, often the access to critical functions is restricted to a network party that is encoded in the session identifier. However, it is possible to construct caller-independent functionalities for a large class of UC-functionalities, if the implementation relies on keys but is otherwise stateless. See Section 5 for details.

2.4 Policies

Now that all credentials on different Security APIs in the network are abstracted to a central storage, we can describe the implementation of a global policy. Every credential in $\mathcal{F}^{\mathrm{KM}}$ has exactly one attribute from a set of attributes A.

Definition 5 (Policy). *Given the KU-functionalities* $\mathcal{F}_i$*,* $i \in \{1, \ldots, l\}$ *and corresponding sets of commands* $\mathcal{C}_i$*, the* policy *is a quaternary relation* $\Pi \subset \{\mathcal{F}_1, \ldots, \mathcal{F}_l, \mathcal{F}^{\mathrm{KW}}\} \times \cup_{i \in \{1,\ldots,l\}} \mathcal{C}_i \cup \{\texttt{New}, \texttt{Wrap}, \texttt{Unwrap}, \texttt{AttributeChange}\} \times A \times A$ *such that if* $(\mathcal{F}, C, a, a') \in \Pi$ *and*

- $C = \texttt{New}$*, then* $\mathcal{F}^{\mathrm{KM}}$ *allows the creation of a new key for the functionality* $\mathcal{F}$ *with attribute* a*.*
- $\mathcal{F} = \mathcal{F}_i$ *and* $C \in \mathcal{C}_i$*, then* $\mathcal{F}^{\mathrm{KM}}$ *will accept sending the command* C *to* $\mathcal{F}$*, if the handle used is of type* $\mathcal{F}$ *and has the attribute* a*.*
- $\mathcal{F} = \mathcal{F}^{\mathrm{KW}}$ *and* $C = \texttt{Wrap}$*, then* $\mathcal{F}^{\mathrm{KM}}$ *allows the wrapping a key with attribute* a' *using a wrapping key with attribute* a*.*
- $\mathcal{F} = \mathcal{F}^{\mathrm{KW}}$ *and* $C = \texttt{Unwrap}$*, then* $\mathcal{F}^{\mathrm{KM}}$ *allows the unwrapping of a wrapping annotated with the attribute* a' *using a wrapping key with attribute* a*.*

– *if* $C =$ `AttributeChange`*, then* $\mathcal{F}^{\mathrm{KM}}$ *allows the changing of a key's attribute from* a *to* a'*.*

The existence of the latter command implies that a key can have different attributes set for different users of $\mathcal{F}^{\mathrm{KM}}$, corresponding to different Security APIs in the real word. Note that a' is only relevant for the `AttributeChange` command.

Therefore, `attr` in $\mathcal{F}^{\mathrm{KM}}$ maps a user and a handle (U, h) to an attribute, rather than a key-index $i = I(U, h)$.

The policy relation allows us to state exactly how attributes can be changed and what implications they have. It is sufficiently expressive for many policies employed in practice. In Section 3, we will consider a static key-hierarchy as an example. Note that whether $\mathcal{F}^{\mathrm{KM}}$ and its KU-functionalities can be UC-realized depends both on the policy and the functionalities.

2.5 The Key Management Functionality $\mathcal{F}^{\mathrm{KM}}$

A detailed description of $\mathcal{F}^{\mathrm{KM}}$ is given in Listing 1.1. By convention, the response to a query (Command, *sid*, ...) is always of the form (Command$^\bullet$, *sid*, ...), or $\perp$. One may note that at any time the attacker may send queries on behalf of any corrupted user.

The principle structure of $\mathcal{F}^{\mathrm{KM}}$ is that of a proxy service to the KU-functionalities. It is possible to create keys (Generate,...), which means that $\mathcal{F}^{\mathrm{KM}}$ asks the KU-functionality for the credentials and stores them, but outputs only a handle referring to the key. When a command is $C \in \mathcal{C}_i$ is called with a handle and a message, $\mathcal{F}^{\mathrm{KM}}$ substitutes the handle with the associated credential, and forwards the output to $\mathcal{F}_i$, but only if this is in accordance with the policy. The response from $\mathcal{F}_i$ is forwarded unaltered.

The commands that are important for key-management are handled by $\mathcal{F}^{\mathrm{KM}}$ itself. It is possible to wrap and unwrap a key. Here, $\mathcal{F}^{\mathrm{KM}}$ makes use of the key-wrapping functionality $\mathcal{F}^{\mathrm{KW}}$ discussed in Section A (see Listing 1.8 on page 27 for its definition). Note that the user can choose an identifier that is bound to a wrapping in order to identify which key was wrapped. This could, e. g., be a key digest provided by the KU-functionality the key belongs to. If not given explicitly, we assume that $\mathcal{F}^{\mathrm{KW}}$ only leaks the length of the key that is wrapped, i. e., the leakage function $L^a(k_1, k_2)$ returns a bitstring of length $|k_2|$ drawn from a uniform random distribution for each a, k_1 and k_2.

When a wrapped key is unwrapped, $\mathcal{F}^{\mathrm{KM}}$ checks if this key is in its database already. If this is the case, a new handle h' for the user U' is created and mapped to the same index i that references the credentials and the type of the key. Since it is possible to change a key's attributes using the command (AttributeChange), the pair (U', h') identifies the key's attribute, instead of the index i.

Key-management is of limited interest without pre-shared keys as it would not allow us to migrate keys from one device to another. Therefore we model a setup phase: *Room* is a list of users in a secure environment that are allowed to share keys during the set-up phase, which means that the implementation is allowed

to use secure channels to transport keys during this phase, but not later. On a physical device this would correspond to pre-installed keys at manufacturing time or installation of keys by an administrator using a trusted machine. Once the setup phase is finished the functionality enters the run phase. During this phase users may create new keys, wrap and unwrap keys, change their attributes and send commands to the key usage functionalities. Note that keys are identified by a pair (user, handle) and all queries are checked against the policy. The adversary $\mathcal{A}$ may corrupt handles. The UC framework does not provide a global register for corrupted keys, so to avoid the simulator corrupting handles the adversary did not corrupt unnoticed, the environment can check the corruption status of a handle. Before defining $\mathcal{F}^{\mathrm{KM}}$, we will define the relevant parameters describing the network to be modelled.

Definition 6 (Parameters to a Security Token Network). *We summarize the parameters of a Security Token Network(cf. 9) as a tuple* $(\Phi, \Phi^{\mathrm{ext}}, \mathit{Room}, \overline{\mathcal{F}}, \Pi)$, *where*

- $\Phi = \{\varphi_1, \ldots, \varphi_m\}$ *are the dummy users with network identities* $U_1, \ldots, U_m$,
- $\Phi^{\mathrm{ext}} = \{\varphi_{m+1}, \ldots, \varphi_{m+n}\}$ *are the external dummy users with identities* $E_1, \ldots, E_n$,
- $\mathit{Room} \subset \Phi$,
- $\overline{\mathcal{F}} = \{(\mathcal{F}_1, \mathcal{C}_1), \ldots, (\mathcal{F}_l, \mathcal{C}_l)\}$ *are key-manageable functionalities with corresponding sets of commands, and* $\mathcal{F}^{\mathrm{KM}} \notin \{\mathcal{F}_1, \ldots, \mathcal{F}_l\}$,
- *and* Π *is a policy for* $\overline{\mathcal{F}}$

Definition 7 ($\mathcal{F}^{\mathrm{KM}}$). *With respect to the parameters* $(\Phi, \Phi^{\mathrm{ext}}, \mathit{Room}, \overline{\mathcal{F}}, \Pi)$, $\mathcal{F}^{\mathrm{KM}}$ *is defined in the following listing. In addition, whenever a lookup of* I *fails, it responds by sending* $\perp$ *to the querier.* $\mathcal{F}^{\mathrm{KM}}$ *has one thread per dummy user in* Φ.

***structure**:*
  $I : (u, h) \mapsto$ *index #functions mapping user and handle to index*
  type(i) *# maps an index to a functionality*
  credentials(i) *#credentials to access functionality*
  attr(u, h) *#attribute that policy assigns to a user's handle*
  List of Corruption requests
In all phases:
***rcv** (*$m_1, \ldots, m_k$*, U) **from** $\mathcal{A}$:*
  ***if** U has been corrupted before*
    proceed like ***rcv** (*$m_1 \ldots m_k$*) **from** U*
***phase** setup:*
***rcv** (New,sid,F,a) **from** any* $U \in \mathit{Room}$
  ***if**((F,New,a,*$*$*)*$\in \Pi$*)*
    create fresh h,i; update I ***such that*** $I(U, h) \mapsto i$
    ***snd** (*Generate*,sid) **to** F; **rcv** (*Generate$^\bullet$*,sid,s) **from** F*
    ***save** credentials (i):=s, attr (U, h):=a, type(i):=F*
    ***snd** (*Generate$^\bullet$*, sid, h) **to** U*
  ***else snd** $\perp$ **to** U*
***rcv** (Share, sid,* (U_1, h_1), (U_2, h_2)*) **from** $U_1 \in \mathit{Room}$*

```
    if I(U_2, h_2) does not exist and U_2 ∈ Room I(U_2, h_2):=I(U_1, h_1)
        snd (Share•,sid) to U_1
    else snd ⊥ to U_1
rcv (FinishSetup) from any U ∈ Room
    send (FinishSetup•,sid) to U and enter phase run
phase run:
rcv (New,sid,F,a) from U ∈ 𝒰 for some F ∈ {ℱ_1, ..., ℱ_l, ℱ^KW}
    :   #behave as in setup phase
rcv (Wrap,sid,h_1,h_2,id) from U:
    if (ℱ^KW, "Wrap",attr(U,h_1),attr(U,h_2))∈ Π
        snd (Wrap,sid,credentials(I(U, h_1)),
        <attr(U,h_2),type(I(U, h_2)),id>,credentials(I(U, h_2))) to ℱ^KW
        rcv (Wrap•, sid, c) from ℱ^KW snd (Wrap•, sid, c) to U
    else snd ⊥ to U
rcv (Unwrap,sid,h_1, w, a, type, id) from U:
    if (ℱ^KW, "Unwrap",attr(U,h_1),a)∈ Π
        snd (Unwrap,sid,credentials(I(U, h_1)),<a,type,id>,w) to ℱ^KW
        rcv (Unwrap•, sid, m) from ℱ^KW
        if i' exists such that type(i')=type and credentials(i')=m
            create fresh h'; I(U,h'):=i'
            If (a ≠ attr(U,h_1)) and
               (type(I(U, h_1)),"AttributeChange",attr(U,h_1),a)∈ Π
               attr(U,h'):=a
        else #corrupted
            create fresh h',i'
            type(i'):=type; credentials(i'):=m; attr(U,h'):=a; I(U,h'):=i'
        snd (Unwrap•, sid, h') to U
    else snd ⊥ to U
rcv (AttributeChange, sid, h, a_new) from U:
    if (type(I(U, h)),''AttributeChange'',attr(U,h),a_new)∈ Π
        attr(U,h):=a_new
    else snd ⊥ to U
rcv (Command,sid,h,m) from U, for Command ∈ 𝒞_i
    if (type(I(U,h))=ℱ_i and (type(I(U,h)), Command, attr(U,h),*)∈ Π)
        snd (Command, sid, credentials(I(U,h)),m) to F_i
        rcv (Command•, sid, m) from F_i; snd (Command•, sid, m) to U
    else snd ⊥ to U
rcv (Corrupt,sid,(U,h)) from 𝒜
    snd (Corrupt•, sid, credentials(i)) to 𝒜; save corruption request for (U, h)
rcv (CorruptionStatus,sid,h) from U
    if corruption request for (U, h) was saved
        snd (CorruptionStatus•,sid,Corrupted) to U
    else snd ⊥ to U
```

Listing 1.1. A key-management functionality $\mathcal{F}^{\mathrm{KM}}$

In standard UC, proving that a network of security tokens UC-realizes $\mathcal{F}^{\mathrm{KM}}$ in the $\mathcal{F}_1, \ldots, \mathcal{F}_l, \mathcal{F}^{\mathrm{KW}}$ would not be strong enough for our purposes, because the environment could not access the KU-functionalities directly. A signature

functionality, for example, might provide an interface to verify signatures to users that do not have a Security Token. Instead, we define a protocol in which those functionalities interact and a number of external users (Φ^{ext}) has direct access to the KU-functionalities. Technically, this protocol is modelled as a network, not as a functionality (although the protocol machines are, in fact, functionalities).

Definition 8 ($\mathcal{F}^{\text{KM}}$-Network). *The network $\mathcal{F}^{\text{KM}}-\text{Net}_{(\Phi,\Phi^{\text{ext}},Room,\overline{\mathcal{F}},\Pi)}$ is the network where all $\phi \in \Phi$ are dummy parties to $\mathcal{F}^{\text{KM}}$ (instantiated for the same parameters), which itself communicates (via subroutine input tapes) with all $\mathcal{F} \in \overline{\mathcal{F}}$ and $\mathcal{F}^{\text{KW}}$. Additionally, every external dummy user $\phi^{\text{ext}} \in \Phi^{\text{ext}}$ allows the environment to adress each $\mathcal{F} \in \overline{\mathcal{F}}$.*

Definition 9 (Security Token Network). *Given an implementation of key-management ST and parameters $(\Phi, \Phi^{\text{ext}}, Room, \overline{\mathcal{F}}, \Pi)$, we define the Security Token Network $\text{STN}_{ST,\Phi,\Phi^{\text{ext}},Room,\overline{\mathcal{F}},\Pi}$ as the network where $m = |\Phi|$ dummy users $\phi \in \{1, \ldots, m\}$ are each connected to a Security Token ST_i, and each ST_i is connected to all $\mathcal{F}_1, \ldots, \mathcal{F}_l$, to $\mathcal{F}^{\text{KW}}$ and to $\mathcal{F}^{\text{SIMT}}$. Additionally, each external dummy user $\varphi^{\text{ext}} \in \Phi^{\text{ext}}$ is connected to each $\mathcal{F} \in \overline{\mathcal{F}}$.*

Definition 10 (Secure Key-Management). *An Security Token ST is secure (under condition C) with respect to $(\Phi, \Phi^{\text{ext}}, Room, \overline{\mathcal{F}}, \Pi)$ iff $\text{STN}_{ST,\Phi,\Phi^{\text{ext}},Room,\overline{\mathcal{F}},\Pi}$ UC-emulates $\mathcal{F}^{\text{KM}}-\text{Net}_{(\Phi,\Phi^{\text{ext}},Room,\overline{\mathcal{F}},\Pi)}$ (under condition C).*

A protocol $\pi^{\mathcal{F}^{\text{KM}}-\text{Net}}$ that calls $\mathcal{F}^{\text{KM}}-\text{Net}_{(\Phi,\Phi^{\text{ext}},Room,\overline{\mathcal{F}},\Pi)}$ is *subroutine respecting*, i. e., the sub-parties of $\mathcal{F}^{\text{KM}}-\text{Net}_{(\Phi,\Phi^{\text{ext}},Room,\overline{\mathcal{F}},\Pi)}$, in particular the KU-functionalities, exchange input/output only with parties (the external dummy users Φ^{ext}) or sub-parties ($\mathcal{F}^{\text{KM}}$) of $\mathcal{F}^{\text{KM}}-\text{Net}$. This is a neccesary condition for the Universal Composition Theorem. In consequence, for very π and ST secure with respect to $(\Phi, \Phi^{\text{ext}}, Room, \overline{\mathcal{F}}, \Pi)$, $\pi^{\mathcal{F}^{\text{KM}}-\text{Net/STN}}$ UC-emulates $\pi^{\mathcal{F}^{\text{KM}}-\text{Net}}$.

3 Realizing key-management for static key-hierarchies

To illustrate our definition and demonstrate its usefulness, we now implement a Security API which allows arbitrary key-usage functionalities and a policy that enforces a hierarchical key-structure. The policy guarantees that corruption of lower-level keys does not impair the security of higher-level keys. We stress that our implementation describes a way to implement a Security API for key-management that is independent of the key-usage functions it provides. Extending $\mathcal{F}^{\text{KM}}$ and the implementation by a new KU-functionality does not require a new proof.

The implementation of a secure token ST is, like $\mathcal{F}^{\text{KM}}$, based on the key-wrapping functionality $\mathcal{F}^{\text{KW}}$. We assume a secure environment for the set-up phase, permitting all users in the set *Room* to generate keys and share them among each others. This is implemented by the functionality $\mathcal{F}^{\text{SIMT}}$, defined in Listing 1.2 below. This secure channel is only used during the set-up phase.

$\mathcal{F}^{\text{SIMT}}$ is designed after $\mathcal{F}^{\text{SIMT}}$ from [Can05], with two differences: *(i)* there is no corruption, because we want to simulate a secure set-up phase, and nothing more; *(ii)* the adversary is not notified when a message is sent.

```
sid = (S, R, sid')
rcv (Send,sid,m) from S
    snd (Sent,sid,m) to R and halt forever
```

Listing 1.2. A Secure Instantenous Message Transmission Funct. $\mathcal{F}^{\text{SIMT}}$

The implementation ST is inspired by the implementation introduced in [KSW11]. There is an improvement that became apparent during the proof of UC emulation (Theorem 12 below). Namely, when unwrapping with a corrupted key, $\mathcal{F}^{\text{KM}}$ checks the attribute that is going to be assigned to the (imported) key against the policy, instead of just accepting that a corrupted wrapping-key might just import any wrapping the attacker generated. This prevents, for example, that a corrupted wrapping-key of low security *creates* a high-security wrapping-key by unwrapping dishonestly produced wrappings. This detail in the definition of $\mathcal{F}^{\text{KM}}$ enforces a stronger implementation than the one in [KSW11]: ST validates the attribute given with a wrapping, enforcing that it is at least sound according to the policy, instead of blindly trusting the authenticity of the wrapping mechanism. Hence our implementation is more robust.

Definition 11 (ST). *Given parameters $(\Phi, \Phi^{\text{ext}}, Room, \overline{\mathcal{F}}, \Pi)$, the ST that is connected to the dummy-user φ_i with identity U_i is defined in the following listing. In addition, whenever a lookup to* `List` *fails, or a response does not match the pattern in* `rcv`*, ST sends $\perp$ to the querier. ST is single-threaded.*

```
structure: List of (h, type(h), attr(h), credentials(h)), K_corrupted ⊂ handles
phase setup:
if U_i ∉ Room
 rcv (Sent, (U, U_i, sid), FinishSetup) from F^SIMT
      enter run
else #if U_i ∈ Room
 rcv (New,sid, F, a) from U_i
     if a=0 and (F=F^PKE or F=F^Nonce) or (a>0 and F=KW)
          generate fresh h
          snd (Generate,sid) to F
          rcv (Generate•,sid,cred) from F
          save (h,F, a, cred)
          snd (Generate•, sid, h) to U_i
 rcv (Share, sid, (U_i, h_1), (U_j, h_2)) from U_i
     if h_1 is in List and U_2 ∈ Room
          snd (Send,(ST_i, ST_2, sid), ((U_j, h_2),type(h_1),attr(h_1),
                                credentials(h_1))) to F^SIMT
          if rcv (Sent,(ST_2, ST_i, sid),⊥) from F^SIMT snd ⊥ to U_i
          else snd (Share•,sid) to U_1
 rcv (Sent, (ST_j, ST_i, sid),((U, h), type, a, cred)) from F^SIMT
     if h is not in List yet
          add (h,type,a,cred) to List
```

```
            snd (Send,(U_i, U_j, sid),OK)
      else snd (Send,(U_i, U_j, sid),⊥) to F^SIMT
 rcv (FinishSetup) from any U
      for any U ∈ Room
            snd (Send,(U_i, U, sid), FinishSetup) to F^SIMT
            rcv (Sent,(U, U_i, sid), FinishSetup•) to F^SIMT
      snd (FinishSetup•,sid) to U
      enter phase run
 rcv (Sent, (U, U_i, sid),FinishSetup) from F^SIMT
      snd (Send,(U_i, U, sid), FinishSetup•) to F^SIMT
      enter phase run
phase run:
rcv(New,sid, F, a) from U_i:

        ⋮     #behave as in setup phase

rcv (AttributeChange,sid,h,a_new) from U_i:
      if attr(h)=a_new: snd (AttributeChange,sid) to U_i
      else snd ⊥ to U_i
rcv (Wrap,sid,h_1, h_2,id) from U_i:
      if attr(h_1)=0 or attr(h_1)≤attr(h_2): snd ⊥
      else # attrh(h_1)≥1 and attr(h_1)>attr(h_2)
            snd (Wrap, sid,sid(h_1),credentials(h_1),
                  <attr(h_2),type(h_2),id>,credentials(h_2)) to F^KW
            rcv (Wrap•,wrap) from F^KW;
            snd (Wrap•,sid,(wrap,<attr(h_2),type(h_2),id>)) to U_i
rcv (Unwrap,sid,h,c, a, F, id)) from U_i:
      if attr(h) ≥ 1 and attr(h) > a
            snd (Unwrap,sid,credentials(h),<a,F,id>,c) to F^KW;
            rcv (Unwrap,sid,k) from F^KW
            generate fresh h'; save (h', F, a, k)
rcv (Command,sid,h,m) from U_i, Command ∈ C_j
      if attr(h)≠0 or h does not exist: snd ⊥ to U_i
      else # attr(h)=0
            snd (Command, sid, credentials(h),m) to F_j
            rcv (Command•, sid, m) from F_j; snd (Command•, sid, m) to U_i
rcv (Corrupt,sid,(U,h)) from A
      if U=U_i
            snd (Corrupt•, sid, credentials(h)) to A; save corruption request for h
rcv (CorruptionStatus,sid,h) from U_i
      if corruption request for h was saved
            snd (CorruptionStatus•,sid,Corrupted) to U
```

Listing 1.3. A security token ST_i using $\mathcal{F}^{\mathrm{KW}}$ and $\mathcal{F}^{\mathrm{SIMT}}$

We now show that ST implements the following *static key-hierarchy policy*. We define this policy as the relation that consists of all 4-tuples ($\mathcal{F}$,Cmd,attr_1,attr_2) such that the conditions in one of the lines in the following table hold. Note that

we omit the "=" sign when we mean equality and "*" denotes that no condition has to hold for the variable.

$\mathcal{F}$	Cmd	attr_1	attr_2
$\mathcal{F}^{\text{KW}}$	New	> 0	*
$\neq \mathcal{F}^{\text{KW}}$	New	0	*
*	AttributeChange	a	a
$\mathcal{F}^{\text{KW}}$	Wrap	> 0	$\text{attr}_1 > \text{attr}_2$
$\mathcal{F}^{\text{KW}}$	Unwrap	> 0	$\text{attr}_1 > \text{attr}_2$
$\mathcal{F}_i$	$C \in \mathcal{C}_i$	0	*

(where $a \in \mathbb{N}$)

Cortier and Steel have shown that a policy enforcing a static key-hierarchy can give the following guarantee: If an attacker leans a key of a certain security level a (the attribute in our case), he might learn any data of security level strictly smaller than a, but nothing more [CS09]. Transferring their result to our framework remains ongoing work.

We can now state our UC-emulation theorem which is depicted in Figure 1.

Theorem 12. *ST is secure with respect to any parameters $(\Phi, \Phi^{\text{ext}}, \mathit{Room}, \overline{\mathcal{F}}, \Pi)$, where Π describes a static-key hierarchy.*

Proof sketch. We prove that an invariant holds for all sequences of messages that the environment sends to the network $\text{STN}_{ST,\Phi,\Phi^{\text{ext}},\mathit{Room},\overline{\mathcal{F}},\Pi}$ as $ST^{\mathcal{F}_1 \dots \mathcal{F}_l}$. Assuming that the invariant is preserved for a smaller sequence of messages, we show that it is also preserved after the next message is processed. The heart of the invariant is the consistency of the internal state of $\mathcal{F}^{\text{KM}}$ and the "global state" of the ST_i, e. g., that for all internal indexes $i = I(U_i, h)$ in $\mathcal{F}^{\text{KM}}$, we have that, on ST_i, $\texttt{credentials}(i)$ and $\texttt{credentials}(h)$ are equally distributed, $\texttt{attr}(i) = \texttt{attr}(h)$, and that $\mathcal{F}^{\text{KM}}$ and all $ST_i \in \mathit{Room}$ leave the set-up phase at the same time. We further need to preserve other properties, for example that the internal states of the functionalities $\mathcal{F}_1 \dots \mathcal{F}_l$ in both worlds are equally distributed and credentials of wrapping keys stay secret. An intuition why state consistency holds is the following: The credentials and attributes are only written by an (*Unwrap*) or by a (`New`) query. (`New`) is trivial to verify, so we look at (*Unwrap*). $\mathcal{F}^{\text{KW}}$ only allows (*Unwrap*, *sid*, h, a, c) if c has been produced by a (*Wrap*, *sid*, h, a, m) query. By the fact that the internal handles of $\mathcal{F}^{\text{KW}}$ cannot be guessed and never leave the device, we can guarantee that query came from an ST_i. This means especially that the bound attribute a is the saved attribute, so we can guarantee $\texttt{attr}(i) = \texttt{attr}(h)$. Furthermore, by the fact that the credentials constitute the message m, we can assure that the credentials are drawn from the same distribution. Intuitively, if we assume the internal state of a KU-functionality to be equally distributed in both worlds, we can conclude from state consistency that every command for this KU-functionality behaves just the same in both worlds, in particular since the credentials are equally distributed. See Appendix B for the full proof. □

The Theorem specifies how to use key-wrapping to implement a key-hierarchy. Due to its generality, it is simple to add new features to a Security API, such as digital signatures, without having to prove anything about the interaction of the new feature with other KU-functionalities. As long as the token's key-management is implemented according to ST, the proof does not need to be redone. In this sense, Theorem 12 allows us to extend a token with arbitrary KU-functionalities - as long as they can be implemented individually, which is the subject of the next section.

We will briefly discuss the runtime of ST in detail. ST maintains a configuration of size $O(\eta \cdot q)$, where q is the number of keys stored, which is at most as large as the number of *New* or *Unwrap* queries received so far. For each query received, the configuration grows at most by the length of one entry in the database, i. e., by $O(\eta)$, performs $O(\eta)$ steps of computation (since a credential of length in $O(\eta)$ might be transmitted) and at most one subroutine-call (i. e.input to another functionality) is made.

Note that the notion of polynomial-boundedness of a protocol does not transfer well from one simulation-based security framework to another. This is polynomial-time execution according to the definition in the IITM framework [Küs06], because the run-time per activation is bounded by a polynomial in the security parameter, the current input size and the size of the current configuration, while the runtime at any point of the execution is bounded by a polynomial in the security parameter, and the length received on the input-channels to the environment (which would be tagged as "enriching" in this framework). In contrast, the definition in the UC framework requires the runtime at any point of the execution to be bounded by a polynomial in n, where n is the sum of the security parameter η, the number bytes input or output to sub-routines , and $n_N \cdot \eta$ (n_N is the number of functionalities called, in our case at most $l+2$). This means that the number of input bytes *per query* needs to be a polynomial in η, since at most one sub-routine output is made per query. This can be reached by altering the definition of $\mathcal{F}^{\mathrm{KM}}$ and ST such that 1^η needs to be appended to every incoming query, but is ignored. In fact, in the GNUC framework, this is the case for every message transmitted [HS11], thus ST runs in polynomial-time according to the definition in this framework.

4 Realizing key-usage functionalities

To demonstrate the usefulness of Theorem 12, we will equip a security token ST as shown above with the functionalities $\mathcal{F}_1 = \mathcal{F}^{\mathrm{Nonce}}$ and $\mathcal{F}_2 = \mathcal{F}^{\mathrm{SIG}}$ defined below. The resulting security token $ST^{\mathcal{F}^{\mathrm{Nonce}},\mathcal{F}^{\mathrm{SIG}}}$ is able to encrypt keys and nonces and sign user-supplied data. It is not able to sign keys, as we consider this task part of the key-management. We suppose that the symbolic results in this paper can be transferred to $ST^{\mathcal{F}^{\mathrm{Nonce}},\mathcal{F}^{\mathrm{SIG}}}$ without much effort, as by Theorem 12, $ST^{\mathcal{F}^{\mathrm{Nonce}},\mathcal{F}^{\mathrm{SIG}}}$ can realize key-management for $\{\mathcal{F}^{\mathrm{Nonce}},\mathcal{F}^{\mathrm{SIG}}\}$ and a static key-hierarchy policy. This allows for an easier analysis in the context of a protocol. Formal methods could be employed: an approach of using abstractions

by UC functionalities to prove computational soundness is presented in [CH06]. It seems promising to derive a computational soundness result for the symbolic model presented in [CS09]. The digital signature functionality is designed after the one described in [KT08] and detailed in Listing 1.4.

params: $L, \mathcal{M}, \mathcal{Y}$ *#Leakage function domain of plaintexts,signatures*
structure: $\mathcal{K}$ *#set of generated keys*
$\mathcal{K}_{\text{corrupted}} \subset \mathcal{K}$, $\mathcal{K}_{\text{uncorrupted}} := \mathcal{K} \setminus \mathcal{K}_{\text{corrupted}}$
Signatures: $\mathcal{K} \to 2^{\mathcal{M} \times \mathcal{Y}}$
List: $\mathcal{P} \times \mathcal{K}$
Algorithms Sig, Ver
phase init:
rcv (Algorithms, *sid*, sig, ver) **from** $\mathcal{A}$
Sig:=sig; Ver:=ver; **enter phase** run
phase run:
rcv (Generate, *sid*) **from** P
snd (Generate, *sid*) **to** $\mathcal{A}$
rcv (KeyGenKey, *sid*, $k := (sk, vk)$) **from** $\mathcal{A}$
$ptr \leftarrow \{0,1\}^{|k|}$
add (ptr, k) **to** List
snd (Generate$^\bullet$, *sid*, *ptr*) **to** P
rcv (VerificationKey,*sid*,ptr)
search $(ptr, (sk, vk)) \in$List
snd (VerificationKey$^\bullet$,*sid*,vk) **to** P
rcv (Sign,*sid*, $ptr \in \mathcal{P}$, $m \in \mathcal{M}$) **from** P
search $(ptr, (sk, vk)) \in$List
$\sigma \leftarrow \text{Sig}(sk, m)$
if $\text{Ver}(vk, m, \sigma)$
save (m, σ) **in** Signatures(vk)
else $\sigma \leftarrow \perp$
snd (Sign$^\bullet$,*sid*,σ) **to** P
rcv(Verify, *sid*, $ptr \in \mathcal{P}$, m, σ) **from** P
search $(ptr, k = (sk, vk)) \in$List
$b \leftarrow \text{Ver}(vk, m, \sigma)$
if $k \notin \mathcal{K}_{\text{corrupted}}$ **and** b=1 **and** $\nexists \sigma' : (m, \sigma') \in$Signatures($vk$)
or $b \notin \{0,1\}$
$b \leftarrow \perp$
snd (Verify$^\bullet$,*sid*,b) **to** P
rcv(VerifyPublic,*sid*,*vk*,*m*,σ)
search $(ptr, k = (sk, vk)) \in$List
if such *ptr* does not exist, **or** $k \in \mathcal{K}_{\text{corrupted}}$
$b \leftarrow \text{Ver}(vk, m, \sigma)$
else
$b \leftarrow \text{Ver}(vk, m, \sigma)$
if b=1 **and** $\nexists \sigma' : (m, \sigma') \in$Signatures($vk$)
or $b \notin \{0,1\}$
$b \leftarrow \perp$
snd (VerifyPublic$^\bullet$,*sid*,b) **to** P
rcv(Corrupt,*sid*,*ptr*) **from** $\mathcal{A}$
if $(ptr, k) \in$List

add k **to** $\mathcal{K}_{\text{corrupted}}$ **and save** corruption request for *ptr*
rcv (CorruptionStatus,*sid*,*ptr*) **from** U_i
if corruption request for *ptr* was saved
snd (CorruptionStatus$^\bullet$,*sid*,Corrupted) **to** U
else snd $\perp$ **to** U

Listing 1.4. A signature functionality $\mathcal{F}^{\text{SIG}}$

Note that the functionality offers a command `VerificationKey` $\in \mathcal{C}_{\text{SIG}}$ that outputs the public part of the key. Effectively, this means that when $\mathcal{F}^{\text{SIG}}$ is used via $\mathcal{F}^{\text{KM}}$ and a user sends this command to $\mathcal{F}^{\text{KM}}$, using a handle that is of type $\mathcal{F}^{\text{SIG}}$, the output contains the verification key. The function `VerifyPublic` then allows to verify the validity of a signature for a given verification key, without knowledge of the credential. In Section 5, we mention that future work could enable us to produce an implementation $\hat{\mathcal{I}}^{\text{SIG}}$ such that $\hat{\mathcal{I}}^{\text{SIG}} \leq_{UC} \mathcal{F}^{\text{SIG}}$ from the proof presented in [KT08] for a non-key-manageable signature functionality. Since this is out of the scope of this work, we assume such a $\hat{\mathcal{I}}^{\text{SIG}}$.

The $\mathcal{F}^{\text{Nonce}}$ is an unusual functionality, but demonstrates what can be done within the design of $\mathcal{F}^{\text{KM}}$. It models the creation of nonces, tests of equality and two kinds of corruption, one allowing him to learn the value of a nonce, and a second one allows him to create a nonce that is equal to every other nonce, even nonces that have not yet been created. $\mathcal{F}^{\text{Nonce}}$ might be used to model nonce corruption through weak random number generators or investigate on nonce generation via PRNGs, for example.

structure: List: $\mathcal{P}$, $\mathcal{P}_{\text{cor}} \subset \mathcal{P}$
rcv(Generate, *sid*) **from** any P:
h $\leftarrow \{0,1\}^\eta$
snd (Generate$^\bullet$, *sid*, h) **to** P
rcv(Equal, *sid*, h_1, h_2) for any P
if $h_1 = h_2$ **or** $h_1 \in \mathcal{P}_{\text{cor}}$ **or** $h_2 \in \mathcal{P}_{\text{cor}}$ then
snd (Equal$^\bullet$,*sid*,YES) **to** P
else
snd (Equal$^\bullet$,*sid*,NO) **to** P
rcv(Corrupt, *sid*,h) **from** $\mathcal{A}$
snd (Corrupt$^\bullet$, *sid*, h) **to** $\mathcal{A}$
save corruption request for h
rcv(GenerateNonceGuess, *sid*) **from** $\mathcal{A}$
h $\leftarrow \{0,1\}^\eta$
add h **to** $\mathcal{P}_{\text{cor}}$
save corruption request for h
snd (GenerateNonceGuess$^\bullet$, *sid*, h) **to** $\mathcal{A}$
rcv (CorruptionStatus,*sid*,*h*) **from** U_i
if corruption request for h was saved
snd (CorruptionStatus$^\bullet$,*sid*,Corrupted) **to** U
else snd $\perp$ **to** U

Listing 1.5. A nonce-generaration functionality $\mathcal{F}^{\text{Nonce}}$

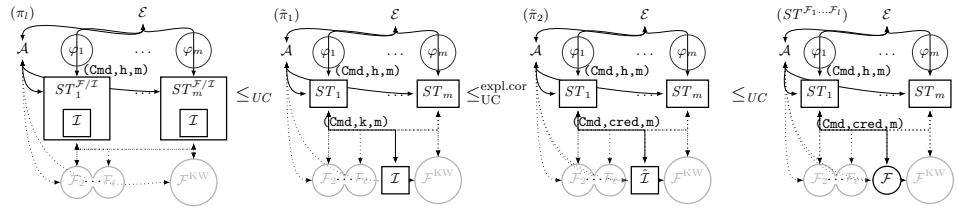

Fig. 2. Simplified sketch of the chain of proofs to show that KU-functionalities can be implemented.

Since it is established that $ST^{\mathcal{F}^{\text{Nonce}},\mathcal{F}^{\text{SIG}}}$ UC-emulates $\mathcal{F}^{\text{KM}}$ for $\mathcal{F}^{\text{Nonce}}$ and $\mathcal{F}^{\text{SIG}}$, it is possible to obtain a number of properties in spirit of those proven for a symbolic modelling of a static key-hierarchy policy in [CS09], since the token $ST^{\mathcal{F}^{\text{Nonce}},\mathcal{F}^{\text{SIG}}}$ provides similar cryptographic operations. Here, however, we investigate how to realize KU-functionalities in general:

A proof technique to realize KU-functionalities Although Theorem 12 offers a great deal of flexibility, the question arises whether the functionalities can be realized in this context. Most notably, ST wraps credentials instead of keys, but the environment sees credentials only in case of corruption. So, is it possible to implement credentials with keys, knowing that the KU-functionalities are in some sense protected by ST? We will introduce a proof technique permitting the substitution of calls to a KU-functionality with a distributed implementation that operates using keys instead of credentials. It is quite generic and allows us to conclude under what circumstances an implementation of key-management and cryptographic operations in a device UC-emulate $\mathcal{F}^{\text{KM}}$ with a given set of KU-functionalities. It is of further interest, because it gives an answer to the question of how to abstract keys in simulation-based security frameworks like UC.

For distributed Security APIs, the implementation of a KU-functionality $\mathcal{F}$ cannot rely on any internal state other than the key. Therefore, we can safely assume that a typical implementation consists of a key-generation algorithm KG and an implementation $impl_{\texttt{Cmd}}$ for each command $\texttt{Cmd}$ in the set of supported commands $\mathcal{C}$. When generating a new key, a Security API creates a key $k \leftarrow \text{KG}(1^\eta)$ and stores it as if it were a credential. When processing a request $(\texttt{Cmd}, sid, h, m)$, assuming that h is a handle to k, the response is computed by $impl_{\texttt{Cmd}}(k, m)$. To model a Secure API that makes this computations, we assume it simulates the following implementation of the functionality locally:

Definition 13 (Local implementation). *For a functionality $\mathcal{F}$ with commands $\mathcal{C}$, given* KG *and $\{impl_{\texttt{Cmd}}\}_{\texttt{Cmd}\in\mathcal{C}}$, we define $\mathcal{I}$ as follows:*

rcv(Generate,*sid*) ***from*** *U*
  snd (Generate$^\bullet$, *sid*, $k \leftarrow \text{KG}(1^\eta)$) ***to*** *U*

rcv *(*Command*,sid,k,m)* ***from*** U , Command $\in \mathcal{C}_j$
    snd *(*Command$^\bullet$*,sid,*$impl_{\texttt{Cmd}}(k,m)$*)* ***to*** U

Listing 1.6. A generic implementation of a key-usage function, $\mathcal{I}$

Given an ITM π, we say that $\pi^{\mathcal{F}/\mathcal{I}}$, i. e., π where every subroutine call to $\mathcal{F}$ is is substituted by a call to a locally simulated instance of $\mathcal{I}$, is π with a local implementation of $\mathcal{F}$ by $\mathcal{I}$.

From its definition we can see that $ST^{\mathcal{F}/\mathcal{I}}$ can be implemented using only function calls to the algorithms KG and $\{impl_{\texttt{Cmd}}\}_{\texttt{Cmd}\in\mathcal{C}}$. It is clear that it is typically impossible to directly UC-emulate $\mathcal{F}$ by $\mathcal{I}$, since $\mathcal{I}$ exposes the key generated by KG to the environment. A slight modification allows us to use credentials instead of keys. Credentials need to be difficult to guess, but since wrappings of credentials will be visible to the environment, we need to assure that the distribution of length of credentials is similar to the distribution of length of keys.

params: KG with domain $\mathcal{K}$,$impl_{\texttt{Cmd}}$ for each $\texttt{Cmd} \in \mathcal{C}$
structure: List: $\mathcal{K} \times \mathcal{K}$, $\mathcal{K}_{\text{corrupted}} \subset \mathcal{K}$
rcv(Generate,*sid*) **from** U
    save ($k \leftarrow$ KG(1^η), *cred* $\leftarrow \{0,1\}^{|k|}$) **in** List;
    snd (Generate$^\bullet$,*sid*,*cred*) **to** U
rcv (Cmd,*sid*,cred,m) **from** U , $\texttt{Cmd} \in \mathcal{C}_j$
    search (k,cred)$\in$ List; **snd** (Cmd$^\bullet$,*sid*,$impl_{\texttt{Cmd}}(k,m)$) **to** U
rcv (Corrupt,*sid*,cred) **from** $\mathcal{A}$
    search (k,cred)$\in$ List; **add** k **to** $\mathcal{K}_{\text{corrupted}}$; **snd** (Corrupt$^\bullet$,*sid*,$k$) **to** $\mathcal{A}$
rcv (CorruptionStatus,*sid*,cred) **from** U
    search (k,cred)$\in$ List
    if $k \in \mathcal{K}_{\text{corrupted}}$: **snd** (CorruptionStatus$^\bullet$,*sid*,Corrupted) **to** P
    else snd (CorruptionStatus$^\bullet$,*sid*,Uncorrupted) **to** P

Listing 1.7. Implementation of KU-function $\hat{\mathcal{I}}$

In the following, we will prove that if $\hat{\mathcal{I}}$ implements a KU-functionality $\mathcal{F}$, then we can substitute calls to $\mathcal{F}$ by local computations as defined by $\mathcal{I}$. In a way, this is a reversed version of a joint state theorem, because we implement a functionality with an internal state by distributing the computation across the ST_is. The proof will proceed in three steps which are depicted in Figure 2. Despite having a globally readable register for which parties are corrupted, the UC framework has nothing of the sort for corrupted keys. This is the reason why dealing with key-corruption within UC is rather quirky: Usually, the implementation offers a (Corrupt) query that exposes a key and can only be called by the adversary. The functionality, when receiving the same query, adjusts its behaviour, usually giving the simulator more power, e. g. in the case of encryption, it sends real encryptions instead of fake once, accounting for the knowledge the adversary has obtained. To assure that the simulator corrupts the same keys that the adversary corrupts, a (CorruptionStatus) query allows the environment to see whether a keys was corrupted. If it was not for such a query, many functionalities could be emulated easily with a simulator that corrupts every key from the start.

Although a real Security API would never answer a (Corrupt) query, this method works well to model corruption in case that the corruption of one key is independent of the corruption of another. This is different in our case, here is an example: Assume, a key k_2 is wrapped using k_1 and transferred from Security API A to Security API B. Security API B now generates a key k_3 and wraps it with $k2$, and sends the wrapping to the environment. Now, assume the adversary corrupts the handle (A,1) that points to k_1. In the real world, the attacker learns all the keys k_1, k_2, k_3. Therefore, the corresponding credentials need to be given to the simulator in the ideal world, but this requires registering which key is wrapped with which key, which is impossible to do for this example, since A, B and C do not have a communication channel except for the environment. To overcome the lack of a global key-corruption register, we need to require that the environment corrupts every key explicitly: Before a key is wrapped with a corrupted key, it has to be corrupted itself.

We define this property with respect to the view of the environment E, i. e., the messages $(m_0,\ldots,m_n)$ it sends or receives. Let $E^t = (m_0, \ldots, m_t)$ denote the t-prefix of E. Since the environment never addresses a party directly, we define $\mathit{from}(m)$ to be the identity of the party that sent the message either herself, or was instructed by the environment to do so. $\mathit{to}(m)$ is defined similarly. Now it is possible to define the response to the ith message in E as $\mathit{response}_{E^t}(i) = \min_{j>i}\{\mathit{from}(m_i) = \mathit{to}(m_j) \wedge \mathit{from}(m_j) = \mathit{to}(m_i) \wedge \nexists k < i : \mathit{response}(k) = i\}$. This assumes that there is a response to every query, i. e., in case of an error, $\mathcal{F}^{\mathrm{KM}}$ responds with $\perp$ rather than ignoring the query. In order to tell which handles are corrupted, we need to define which handles point to the same key a given moment t. Given $E^n = (m_0, \ldots, m_n)$ we define $\equiv^0$ to be the empty relation and for all $1 \leq t \leq n$, we define $\equiv^t$ as the least symmetric transitive relation such that

1. $\equiv^t \subset \equiv^{t-1} \cup\{(U,h),(U,h)\}$, if $m_t = (\mathrm{Generate}^\bullet, \mathit{sid}, \mathrm{h})$, $\mathit{to}(m_t) = U$ and $\exists s < t : m_s = (\mathtt{New}, \mathit{sid}, F, a, h) \wedge \mathit{from}(m_s) = U \wedge \mathit{response}_{E^t}(s) = t$ (for any F and a)
2. $\equiv^t \subset \equiv^{t-1} \cup \{(U_1,h_1),(U_2,h_2)\}$, if $m_t = (\mathrm{Share}^\bullet, \mathit{sid})$, $\mathit{to} = U_1$ and $\exists s < t :$ $m_s = (\mathrm{Share}, \mathit{sid}, (U_1,h_1),(U_2,h_2)) \wedge \mathit{from}(m_s) = U_1 \wedge \mathit{response}_{E^t}(s) = t$
3. $\equiv^t \subset \equiv^{t-1} \cup\{(U_1,h_1),(U_2,h_2)\}$, if $m_t = (\mathrm{Unwrap}^\bullet, \mathit{sid}, h_2)$, $\mathit{to}(m_t) = U_2$ and $\exists q, r, s : \mathit{response}_{E^t}(q) = r$, $\mathit{response}_{E^t}(s) = t$, $r < s$, $m_q = (\mathit{Wrap}, \mathit{sid}, h_1, h_2, \mathrm{id})$, with $\mathit{from}(m_q) = U_1$, $m_r = (\mathit{Wrap}^\bullet, \mathit{sid}, w)$, with $\mathit{to}(m_r) = U_1$ and $m_s = (\mathit{Unwrap}, \mathit{sid}, h_1', w, a, \mathrm{type}, \mathrm{id})$, $\mathit{from}(m_s) = U_1$ where $(U_2, h_1') \equiv^{t-1} (U_1, h_1)$.
4. $\equiv^t = \equiv^{t-1}$, otherwise.

By definition of $\mathcal{F}^{\mathrm{KM}}$, during any execution of $\mathcal{F}^{\mathrm{KM}}$ and $\mathcal{F}^{\mathrm{KW}}$ with a view E^t, $I(U,h) = I(U'h')$ in $\mathcal{F}^{\mathrm{KM}}$ iff $(U,h) \equiv^t (U',h')$. Using this relation, we can define a corrupt handle predicate: $\mathit{corrupted}_{E^t}(U,h)$ iff either some (U^*,h^*), $((U^*,h^*) \equiv^t (U,h))$ were corrupted directly, i. e., there are $m_i, m_j \in E^t$ such that $\mathit{response}\ (i) = j$, $\mathit{from}\ (m_i) = \mathcal{A}$, and $m_i = (\mathrm{Corrupt}, \mathit{sid}, (U^*,h^*))$ and $m_j = (\mathrm{Corrupt}^\bullet, \mathit{sid}, c)$ for some c, or via wrapping with a corrupted key, formally:

there are $e_i, m_j \in E^t$ such that *response* $(i) = j$, *from* $(m_i) = U_1$, and $m_i = (\mathit{Wrap}, \mathit{sid}, h_1, h_2)$ and $m_j = (\mathit{Wrap}^\bullet, \mathit{sid}, c)$ for some c, while $(U_1, h_1) \equiv^t (U^*, h^*)$ and $(U_1, h_2) \equiv^t (U, h)$ for some (U^*, h^*) that were corrupted directly. We define the event *explicit-corruption* to be the event that the view $E^t = (m_0, \dots m_t)$ of the environment at the end of the execution fulfils the following predicate:

$$\begin{aligned}\forall i \leq t.\ m_i = (\mathit{Wrap}, \mathit{sid}, h_1, h_2, \mathrm{id}) \wedge \mathit{from}(m_i) = U \\ \wedge\ \mathit{corrupted}_{E^i}(U, h_1) \Rightarrow \mathit{corrupted}_{E^i}(U, h_2)\end{aligned}$$

Intuitively, this predicate holds if the attacker never uses a corrupt key to wrap an uncorrupted key, i. e., the attacker needs to explicitly corrupt a key in order to wrap it with another corrupted key.

Now we show how to substitute a KU-functionality used of several instances by ST with a local implementation $\mathcal{I}$ running on all ST individually.

Theorem 14. *Given the implementation of keymanagement ST from Definition 11 with parameters* $(\Phi, \Phi^{\mathrm{ext}}, \mathit{Room}, \overline{\mathcal{F}}, \Pi)$*, let* $ST^{\mathcal{F}/\mathcal{I}}$ *be ST with a local implementation of* $\mathcal{F} \in \overline{\mathcal{F}}$ *by* $\mathcal{I}$*. If (i)* $\hat{\mathcal{I}}$ *UC-emulates* $\mathcal{F}$ *and (ii)* $\Pr[k = k' | k' \leftarrow \mathrm{KG}(1^\eta)]$ *is negligible for all* k *in the domain of* KG*, then,* $\mathrm{STN}_{ST,\Phi,\Phi^{\mathrm{ext}},\mathit{Room},\overline{\mathcal{F}}\cup\hat{\mathcal{I}},\Pi}$ *UC-emulates* $\mathrm{STN}_{ST^{\mathcal{F}/\mathcal{I}},\Phi,\Phi^{\mathrm{ext}},\mathit{Room},\overline{\mathcal{F}},\Pi}$ *under condition explicit-corruption.*

Proof. In the following, we will abreviate $\mathrm{STN}_{ST^{\mathcal{F}/\mathcal{I}},\Phi,\Phi^{\mathrm{ext}},\mathit{Room},\overline{\mathcal{F}}\cup\hat{\mathcal{I}},\Pi}$ by π^l, and $\mathrm{STN}_{ST,\Phi,\Phi^{\mathrm{ext}},\mathit{Room},\overline{\mathcal{F}},\Pi}$ by π^r. Note that, in π^l, the external dummy users $\varphi \in \Phi^{\mathrm{ext}}$ have subroutine access to $\hat{\mathcal{I}}$ like to any other KU-functionality.

from π^l *to* $\tilde{\pi}^1$*:* The network $\tilde{\pi}^1$ functions as an intermediate step. It is π^l, but with ST instead of $ST^{\mathcal{F}/\mathit{calI}}$, each adressing the same machine running $\mathcal{I}$ via sub-routine tapes when $\mathcal{F}$ should be adressed. When *Sim* is supposed to address $\mathcal{I}$ for the environment, *Sim* acts as if it were not present, since it cannot be addressed in the real-world, either. Since $\mathcal{I}$ does not have any internal state, there is no difference in the output of $ST_i^{\mathcal{F}/\mathcal{I}}$ simulating $\mathcal{I}$, and ST_i calling the machine $\mathcal{I}$ in the network. Since no other party is able to address $\mathcal{I}$, there is no environment $\mathcal{E}$ that can distinguish between the two networks The second step is more involved:

from $\tilde{\pi}^1$ *to* $\tilde{\pi}^2$ In this transition, we will abstract keys by credentials. $\tilde{\pi}^2$ is defined like $\tilde{\pi}^1$, but ST accesses $\hat{\mathcal{I}}$ instead of $\mathcal{I}$, just like the external dummy users.On that account, we define *Sim* as follows: It runs $\mathcal{D}$ with the following exception. When *Sim* is instructed to send (Corrupt,*sid*,*h*) to some ST_i, it first requests the corresponding credential, i. e., sends (Corrupt,*sid*,*h*) to ST_i. It obtains *cred* from ST_i. Then, *Sim* sends (Corrupt,*sid*,*cred*) to $\hat{\mathcal{I}}$, to obtain the response (Corrupt$^\bullet$, *sid*, k) and forwards it to $\mathcal{E}$. *Sim* adds (cred, k) to a list of corrupted credentials and corresponding keys. When $\mathcal{E}$ instructs *Sim* to send a (CorruptionStatus) command to $\hat{\mathcal{I}}$, it ignores this message, as $\mathcal{A}$ does not answer this response in the real world. When $\mathcal{F}^{\mathrm{KW}}$ sends the query (*Wrap*,*sid*,*ptr*,*k*,*a*,*cred*) to *Sim*, *Sim* substitutes *cred* by a k that was recorded as (cred, k) while processing a corruption request. It sends (*Wrap*,*sid*,*ptr*,*k*,*a*, k) to the environment and forwards the response back to $\mathcal{F}^{\mathrm{KW}}$.

In order to show that for all $\mathcal{E}$, $\text{EXEC}_{\mathcal{E},\mathcal{D},\tilde{\pi}^1}$ cannot be distinguished from $\text{EXEC}_{\mathcal{E},Sim,\tilde{\pi}^2}$, we will show that throughout the execution, $\tilde{\pi}^2$ and *Sim* provide a perfect simulation of $\tilde{\pi}^1$ and $\mathcal{D}$, as long as the mapping from credentials to keys in $\hat{\mathcal{I}}$ is a bijection. Assuming that this is the case, we note that the only output that depends on some $\texttt{credentials}(h^*)$ and is visible to the environment is the response to *(a)* a (*Wrap*,*sid*,h_1,h^*,*id*) query, *(b)* a (Command,*sid*,h^*,m) query, and *(c)* a corruption request (Corrupt,*sid*,h^*) by the adversary. In case *(a)* we know that h_1 is a handle to a wrapping key, as otherwise ST would not respond in either execution. The output is a wrapping that, by definition of $\mathcal{F}^{\text{KW}}$, contains a random string of the length as $\texttt{credentials}(h^*)$, in case $\texttt{credentials}(h_1)$ are in $\mathcal{K}_{\text{uncorrupted}}$ in $\mathcal{F}^{\text{KW}}$. Both credential and key are of the same length (by definition of $\hat{\mathcal{I}}$), so the output is equally distributed. It is not possible to exploit that $\texttt{credentials}(h^*)$ is now stored in Encryptions(k) in $\mathcal{F}^{\text{KW}}$, since the (*Unwrap*) query that $\mathcal{F}^{\text{KW}}$ provided can not be accessed directly, unless $\texttt{credentials}(h_1)$ was corrupted. (Recall that, by the definition of $\mathcal{F}^{\text{KM}}$, $\mathcal{F}^{\text{KW}} \notin \{\mathcal{F}_1, \ldots, \mathcal{F}_l\}$.) So, if it is called via (*Unwrap*), no output is produced, the credentials are stored instead.

In case that $\texttt{credentials}(h_1)$ is in $\mathcal{K}_{\text{corrupted}}$ in $\mathcal{F}^{\text{KW}}$, we know that the environment must have asked some U to corrupt a handle that points to this credential, as the credential is needed to corrupt a key in $\mathcal{F}^{\text{KW}}$, and the credentials to $\mathcal{F}^{\text{KW}}$ are difficult to guess. Therefore, since we regard only executions that fulfil the conditions of *explicit-corruption*, some $(U', h') \equiv (U, h^*)$ must have been corrupted explicitly. Hence, *Sim* has an entry $(cred, k)$ such that k is the key matching the credential for h^*. *Sim* forwards (*Wrap*,*sid*,*ptr*,k,a,k) for the correct k to the environment, just a $\mathcal{D}$ does in $\hat{\pi}_1$. A correct simulation in case *(b)* is assured by the fact that $\hat{\mathcal{I}}$ substitutes the credential with the corresponding key before executing $impl_{\text{Command}}$, assuming that the List maintained in $\hat{\mathcal{I}}$ describes a bijection between credentials and keys. Therefore, in both executions, the output is $impl_{\text{Command}}(k, m)$, where k is the key that was drawn using KG and bound to the credential. Case *(c)* works similarly: Assuming the bijection, by definition of *Sim*, the environment will receive the key that was created along with the credential. By the assumption that $\Pr[k = k' | k' \leftarrow \text{KG}(1^\eta)]$ is negligible for all k, we know that the probability that every k and *cred* generated in $\hat{\mathcal{I}}$ is distinct, because the length of this list is bound by the running time of the environment, which is polynomial in η. Thus, the mapping from credentials to keys constitutes a bijection. We note further that the external dummy users in Φ^{ext} can adress the same parties as before, but now $\hat{\mathcal{I}}$ stores, in addition to the credentials and keys the dummy user generate, the credentials and keys ST generates. Again by the assumption that $\Pr[k = k' | k' \leftarrow \text{KG}(1^\eta)]$, we see the probability of an interference of those two groups of parties in $\hat{\mathcal{I}}$ are negligible. This concludes the proof that $\text{EXEC}_{\mathcal{E},\mathcal{D},\tilde{\pi}^1}$ can only be distinguished from $\text{EXEC}_{\mathcal{E},Sim,\tilde{\pi}^2}$ with negligible probability.

from $\tilde{\pi}^2$ *to* $ST^{\mathcal{F}_1 \ldots \mathcal{F}_l}$ This follows from the composition theorem provided by the UC framework, and the assumption that $\hat{\mathcal{I}}$ UC-emulates $\mathcal{F}$. (Note that $\mathcal{I}$ and $\mathcal{F}$ are subroutine respecting, since they do not adress any sub-parties.

To conclude, we summarize in Corollary 15 which assumptions have to be met by an implementation of ST, $\mathcal{F}^{\mathrm{KW}}$ and the KU-functionalities in order to form a fully functional Security API that UC-emulates $\mathcal{F}^{\mathrm{KM}}$.

Corollary 15. *A local implementation* $ST^{\mathcal{F}_1/\mathcal{I}_1,\ldots,\mathcal{F}_l/\mathcal{I}_l,\mathcal{F}^{\mathrm{KW}}/\pi^{KW}}$ *of the Security Token described in Definition 11, is secure under condition explicit-corruption* $\cup$ *corrupt-before-wrap, with respect to any parameters* $(\Phi, \Phi^{\mathrm{ext}}, Room, \mathcal{F}_1, \ldots, \mathcal{F}_l, \Pi)$*, if:*

- Π *describes a static-key hierarchy,*
- *for* $i \in \mathcal{F}_1, \ldots, \mathcal{F}_l$, $\tilde{\mathcal{I}}_i$ *UC-emulates* $\mathcal{F}_i$*, and* $\Pr[k = k' | k' \leftarrow \mathrm{KG}(1^\eta)]$ *is negligible for all* k *in the domain of* KG*, the key-generation used in* $\mathcal{I}_i$*,*
- *during the set-up phase, there is a secure channel to realize the secure instantaneous message transfer functionality* $\mathcal{F}^{\mathrm{SIMT}}$*,*
- KW *is a symmetric encryption scheme secure with respect to Definition 18 .*

Proof. Because of lack of space we only sketch this proof which is basically a consequence of Theorems 12, 14 and Theorem 16. Let the network π^l consist of the functionalities $\tilde{\mathcal{F}}$ key-wrapping functionality $\mathcal{F}^{\mathrm{KW}}$ and $U_i \in \mathcal{U}$, each connected to a $ST_i^{\mathcal{F}_1/\mathcal{I}_1,\ldots,\mathcal{F}_l/\mathcal{I}_l}$ with U_i and *Room* as parameters. Then, from the transitivity of UC-emulation, Theorem 12 and by inductive application of Theorem 14, it follows that the network π^l UC-emulates $\{\mathcal{F}^{\mathrm{KM}}, \mathcal{F}^{\mathrm{KW}}, \mathcal{F}_1 \ldots \mathcal{F}_l\}$. Carefully verifying that, by definition of ST and *corrupt-before-wrap*, the composition theorem can be applied, we substitute $\mathcal{F}^{\mathrm{KW}}$ with $\hat{\pi}^{KW}$. Finally, the proof that $\hat{\pi}^{KW}$ can be locally implemented by π^{KW} according to Definition 1.6 is exactly the proof to Theorem 14, except that it is not necessary anymore to deal with case *(a)* in the second step ($\hat{\pi}^1$ to $\hat{\pi}^2$).

5 Conclusions

We have presented a provably secure framework for key management in the UC model. It would be interesting to extend $\mathcal{F}^{\mathrm{KM}}$ allow policies to have side-effects. For example, the attribute of a wrapping key or the pay-load key might be altered when wrapping, so a high-security wrapping key could only be used so many times. A number of policies could be implemented to see what security properties they induced and who much they restrict the use of a Security API. In further work, we are currently also developing a technique for transforming functionalities that use keys but are not key-manageable into key-manageable functionalities with implementations in the style of Listing 1.7. This way, existing proofs could be used to develop a secure implementation of cryptographic primitives in a plug-and-play manner. Investigating the restrictions of this approach could teach us more about the modelling of keys in simulation-based security.

References

BA01. M. Bond and R. Anderson. API level attacks on embedded systems. *IEEE Computer Magazine*, pages 67–75, October 2001.

BCFS10. Matteo Bortolozzo, Matteo Centenaro, Riccardo Focardi, and Graham Steel. Attacking and fixing PKCS#11 security tokens. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, Chicago, Illinois, USA, October 2010. ACM Press. To appear.

BDHK07. Michael Backes, Markus Dürmuth, Dennis Hofheinz, and Ralf Küsters. Conditional reactive simulatability. *International Journal of Information Security (IJIS)*, 2007.

Can01. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145, Las Vegas, Nevada, USA, October 2001. IEEE Computer Society Press.

Can05. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, December 2005. Updated version of [Can01] `http://eprint.iacr.org/`.

CC09. C. Cachin and N. Chandran. A secure cryptographic token interface. In *Proc. 22th IEEE Computer Security Foundation Symposium (CSF'09)*, pages 141–153. IEEE Comp. Soc. Press, 2009.

CCA06. *CCA Basic Services Reference and Guide*, October 2006. Available online at `http://www-03.ibm.com/security/cryptocards/pdfs/bs327.pdf`.

CDNO97. Ran Canetti, Cynthia Dwork, Moni Naor, and Rafail Ostrovsky. Deniable encryption. In Burton S. Kaliski Jr., editor, *CRYPTO*, volume 1294 of *Lecture Notes in Computer Science*, pages 90–104. Springer, 1997.

CH06. Ran Canetti and Jonathan Herzog. Universally composable symbolic analysis of mutual authentication and key exchange protocols. In *In Proceedings, Theory of Cryptography Conference (TCC*, pages 380–403. Springer, 2006.

CHK04. Ran Canetti, Shai Halevi, and Jonathan Katz. Adaptively-secure, non-interactive public-key encryption. Cryptology ePrint Archive, Report 2004/317, 2004. `http://eprint.iacr.org/`.

CKS07. V. Cortier, G. Keighren, and G. Steel. Automatic analysis of the security of XOR-based key management schemes. In O. Grumberg and M. Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, number 4424 in LNCS, pages 538–552, 2007.

CS09. Véronique Cortier and Graham Steel. A generic security API for symmetric key management on cryptographic devices. In *Proc. 14th European Symposium on Research in Computer Security (ESORICS'09)*, volume 5789 of *LNCS*, pages 605–620. Springer, 2009.

DKS10. Stéphanie Delaune, Steve Kremer, and Graham Steel. Formal analysis of PKCS#11 and proprietary extensions. *Journal of Computer Security*, 18(6):1211–1245, November 2010.

DN00. Ivan B. Damgård and Jesper Buus Nielsen. Improved non-committing encryption schemes based on a general complexity assumption. pages 433–451, 2000. RS-00-6.

Hof08. Dennis Hofheinz. Possibility and impossibility results for selective decommitments. Cryptology ePrint Archive, Report 2008/168, 2008. `http://eprint.iacr.org/`.

HS11. Dennis Hofheinz and Victor Shoup. Gnuc: A new universal composability framework. Cryptology ePrint Archive, Report 2011/303, 2011. `http://eprint.iacr.org/`.

KKS12. Steve Kremer, Robert Künnemann, and Graham Steel. Universally composable keymanagement (full version). available online: `http://internet.fr/datei.txt`, 2012.

KSW11. Steve Kremer, Graham Steel, and Bogdan Warinschi. Security for key management interfaces. In *Proceedings of the 24th IEEE Computer Security Foundations Symposium (CSF'11)*, pages 66–82. IEEE Comp. Soc. Press, 2011.

KT08. Ralf Küsters and Max Tuengerthal. Joint State Theorems for Public-Key Encryption and Digitial Signature Functionalities with Local Computation. In *Proc. 21st IEEE Computer Security Foundations Symposium (CSF'08)*, pages 270–284. IEEE Comp. Soc. Press, 2008.

KT11. Ralf Küsters and Max Tuengerthal. Ideal Key Derivation and Encryption in Simulation-Based Security. In *Topics in Cryptology - CT-RSA'11*, volume 6558 of *LNCS*, pages 161–179. Springer, 2011.

Küs06. R. Küsters. Simulation-Based Security with Inexhaustible Interactive Turing Machines. In *Proc. 19th IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 309–320. IEEE Comp. Soc. Press, 2006.

LR92. D. Longley and S. Rigby. An automatic search for security flaws in key management schemes. *Computers and Security*, 11(1):75–89, March 1992.

MR11. Ueli Maurer and Renato Renner. Abstract cryptography. In *Proc. 2nd Symposium in Innovations in Computer Science (ICS'11)*, pages 1–21. Tsinghua University Press, 2011.

RS06. P. Rogaway and T. Shrimpton. Deterministic authenticated encryption: A provable-security treatment of the keywrap problem, 2006.

RSA04. RSA Security Inc., v2.20. *PKCS #11: Cryptographic Token Interface Standard*, June 2004.

TCG07. Trusted Computing Group. TPM Specification version 1.2. Parts 1–3, revision 103. `http://www.trustedcomputinggroup.org/resources/tpm_main_specification`, 2007.

Tse07. Yuh-Min Tseng. An efficient two-party identity-based key exchange protocol. *Informatica*, 18:125–136, January 2007.

A $\mathcal{F}^{\mathrm{KW}}$ and the commitment problem

In $\mathcal{F}^{\mathrm{KM}}$, as well as in the proposed implementation of a Security API, ST, we rely on the key-wrapping functionality $\mathcal{F}^{\mathrm{KW}}$. Due to the commitment problem, it is not possible to realize this functionality under dynamic corruption of keys, therefore $\mathcal{F}^{\mathrm{KM}}$ is not realizable in this form. This following section will explain the commitment problem and explain out the difference of dynamic corruption in the context of Security APIs to the traditional understanding. It will furthermore show how to realize $\mathcal{F}^{\mathrm{KW}}$ using an existing definition of wrapping schemes from [KSW11] for all runs where the commitment problem does not occur.

We build on a definition for deterministic, authenticated wrapping-schemes, going back to the original work by Rogaway and Shrimpton [RS06] which has been adapted to a multi-user setting and to allow key-dependant messages in [KSW11]. We further generalised it to allow the same key to be encrypted several times with different attributes. This is necessary in our work, because we support attribute changes, therefore a key can have different attributes on different machines. A leakage function serves as a parameter in $\mathcal{F}^{\mathrm{KW}}$. It determines the amount of information about the message that is ideally given to the wrapping algorithm.

Typically, it outputs a random string with a length that depends on the length of the message, the key, and the bound attribute. $\mathcal{F}^{\mathrm{KW}}$ is detailed in Listing 1.8.

We chose deterministic encryption because it is used in most hardware tokens, and because we can show that it is possible to implement $\mathcal{F}^{\mathrm{KW}}$ using a key-wrapping scheme as described in Definition 17 in [KSW11]. The security $\mathcal{F}^{\mathrm{KM}}$ provides depends on the authenticity and secrecy provided by $\mathcal{F}^{\mathrm{KW}}$, however, $\mathcal{F}^{\mathrm{KW}}$ can easily be modified to define probabilistic encryption, by not performing the test ''If there is c*'' after reception of a message (Wrap,*sid*,...), but jumping directly into the else branch. It would be interesting to investigate how asymmetric encryption schemes could be used here, but this is out of the scope of this work.

params: $\mathcal{X}$,$\mathcal{K} \subset \mathcal{X}$,$\mathcal{H}, \mathcal{Y}$ *#message−,key−,attribute− and ciphertext space*
$\mathcal{P}$ *#distribution of pointers, with support disjoint from* $\mathcal{X}$
#requirement: $\forall p \in support(\mathcal{P}) : \Pr[p \leftarrow \mathcal{P}]$ *is negligible*
$L : \mathcal{H} \times \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ *#Leakage function*
structure: $\mathcal{K}_{\text{corrupted}} \subset \mathcal{K}$, $\mathcal{K}_{\text{uncorrupted}} := \mathcal{K} \setminus \mathcal{K}_{\text{corrupted}}$
List : $\mathcal{P} \times \mathcal{K}$
Encryptions: $\mathcal{K} \to 2^{(\mathcal{H} \times \mathcal{M} \times \mathcal{Y})}$ *#Encryptions under key k*
Algorithms: $Wrap^{\cdot}(\cdot,\cdot) : \mathcal{H} \times \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$, $Unwrap^{\cdot}(\cdot,\cdot) : \mathcal{H} \times \mathcal{K} \times \mathcal{T} \to \mathcal{X}$
corruption graph $(\mathcal{K}, \emptyset)$
The *sid* is checked (as below), but has no **structure**
phase init:
rcv (Algorithms, *sid*, wrap, unwrap) **from** $\mathcal{A}$
Wrap:=wrap; *Unwrap*:=unwrap; **enter phase** run
phase run:
rcv (Generate, *sid*) **from** P
snd (KeyGen, *sid*) **to** $\mathcal{A}$; **rcv** (KeyGenKey, *sid*, $k \in \mathcal{K}$) **from** $\mathcal{A}$
$ptr \leftarrow \mathcal{P}$; **add** (*ptr*,k) **to** List
snd (Generate$^{\bullet}$, *sid*, *ptr*) **to** P
rcv (Wrap,*sid*, $ptr \in \mathcal{P}$,$a \in \mathcal{H}$,$m \in \mathcal{X} \cup \mathcal{P}$) **from** P
search k **such that** (*ptr*,k) **in** List
if there is c^* **such that** (a,m,c^*) **in** Encryptions(k)
snd ($Wrap^{\bullet}$,*sid*, c^*) **to** P
else if k$\in \mathcal{K}_{\text{corrupted}}$
if $m \in \mathcal{P}$ **and** there is k' **such that** (m, k') **in** List
add edge k, k' **in** corruption graph
for all k'' reachable **from** k' **in** corruption graph
add k'' **to** $\mathcal{K}_{\text{corrupted}}$
snd ($Wrap$,*sid*,*ptr*,k,a,k') **to** $\mathcal{A}$
else
snd ($Wrap$,*sid*,*ptr*,k,a,m) **to** $\mathcal{A}$
rcv ($Wrap^{\bullet}$,*sid*,c) **from** $\mathcal{A}$
else # *if* $k \in \mathcal{K}_{\text{uncorrupted}}$
if $m \in \mathcal{P}$
search k' **such that** (ptr', k') **in** List
add edge k, k' **in** corruption graph
c:= $Wrap^a(k, L^a(k, k'))$
else if $m \in \mathcal{X}$
c:= $Wrap^a(k, L^a(k, m))$

```
        save (a,m,c) in Encryptions(k)
        snd (Wrap•,sid, c) to P
rcv (Unwrap,sid,ptr ∈ P,a ∈ H,c ∈ Y) from P
    search k such that (ptr, k) in List
    if k∈ K_uncorrupted
        if there is an m such that (a, m, c) in Encryptions(k)
            snd (Unwrap•,sid, m) to P # either pointer or message
        else snd ⊥ to P
    else #k∈ K_corrupted
        m = Unwrap^a(k, c)
        if there is k' such that (m, k') in List
            snd (Unwrap•,sid, k') to P
        else if m∈ K: snd (Unwrap•,sid, m) to P
rcv (Corrupt,sid,ptr ∈ P) from A
    search k such that (ptr,k) in List
    for all k' reachable from k in corruption graph
        add k' to K_corrupted
    save corruption request for ptr
rcv (CorruptionStatus,sid,ptr ∈ P) from P
    if corruption request for ptr was saved
        snd (CorruptionStatus•,sid,Corrupted) to U
    else snd ⊥ to U
```

Listing 1.8. A key-wrapping Functianality $\mathcal{F}^{\mathrm{KW}}$

In Appendix C we show that a wrapping scheme as described in [KSW11] can be used to realize this functionality, but only when the environment does not corrupt keys which have been used to encrypt before. We avoid the commitment problem by only regarding executions where a key is never corrupted after it has been used to encrypt. Given the view $E^t = (m_0, \ldots, m_t)$ of the environment we define the event *corrupt-before-wrap* to be the following predicate :

$$\begin{aligned} \forall i \leq t.\ \mathit{corrupted}_{E^t}(U,h) \wedge m_i = (\mathit{Wrap}, \mathit{sid}, h, h') \\ \wedge\ \mathit{from}(m_i) = U \Rightarrow \mathit{corrupted}_{E^i}(U,h) \end{aligned}$$

We state the Theorem here. Both the proof and Definitions 17 and 18 can be found in Appendix C.

Theorem 16. *If a symmetric encryption scheme $KW = (\mathsf{KG}, \mathsf{Wrap}, \mathsf{UnWrap})$ is secure with respect to Definition 18, then for the protocol version $\hat{\pi}^{KW}$ (see Definition 17), $\hat{\pi}^{KW}$ UC-emulates $\mathcal{F}^{\mathrm{KW}}$ under the condition corrupt-before-wrap.*

The interesting part of key-management is *dynamic corruption*: We would like to know what guarantees we can give in case an attacker learns one or more keys during the protocol run. As opposed to static corruption, an attacker can corrupt a key after it has been used. What does this mean in case of encryption? For one, it means that the attacker is not able to distinguish ideally encrypted ciphertexts from actual encryptions of messages, hence, that he does not learn anything about past encryptions after corruption of a key. This idea bears resemblance to the notion of *(perfect) forward secrecy*, [Tse07].

However, this property is not needed, in fact, implementations thereof are impractical in the domain of Security APIs. The question of why we are interested in dynamic corruption arises. A small example, the encryption functionality $\mathcal{F}^{\text{toy}}$, illustrates that the modelling of dynamic corruption is of interest, although forward-secrecy is not a security goal. It furthermore shows where encryption in connection with dynamic corruption produces a situation where existing frameworks for simulation-based security prove to be useless - the so-called commitment problem. $\mathcal{F}^{\text{toy}}$ has a master and a slave key. The corruption of the master key leads to the corruption of the slave key, but not vice versa. In reality, the slave key might have been derived from the master key in some deterministic way, maybe using a hash function.

phase init: *#receive algorithms* `enc`, `dec` *and* $k_{\text{master}}, k_{\text{slave}}$ *from* $\mathcal{A}$
phase run:
rcv (Enc, key$\in$ {master, slave}, m) **from** U:
  if corrupt(key), **snd** (Encryption,c=enc($k_{\texttt{key}}$,m)) **to** U
  else snd (Encryption,c=enc($k_{\texttt{key}}$,$0^{|m|}$)) **to** U
  save (m,c)
rcv (Dec, key$\in$ {master, slave}, c) **from** U:
  if corrupt(key), **snd** (Decryption,dec($k_{\texttt{key}}$,c)) **to** U
  else if (m,c) saved
    snd (Decryption,m) **to** U
rcv (Corrupt, key$\in$ {master)
  if key=master, set corrupt(master) **and** corrupt(slave)
  else set corrupt(slave)
[...]

Listing 1.9. A toy example of an encryption system $\mathcal{F}^{\text{toy}}$

Imagine an implementation of $\mathcal{F}^{\text{toy}}$, say π. Let $\mathcal{E}$ send a number of (Enc, k, m_i) queries, and receive something that looks like encryptions. The simulator *Sim* does, by definition of $\mathcal{F}^{\text{toy}}$, not learn the messages m_i. The environment sees a number of encryptions of the null-string. Now, upon corruption of k, the simulator has to produce a key that allows to open the emitted encryptions to the messages m_i. If we assume some bound $b(n)$ on the length of the key, i. e., the cardinality of the key space is at most $2^{b(n)}$. Still, the environment can produce random messages of a total length greater than $b(n)$. Therefore, there is no key that would open the encryptions independent of the value of m_i to the correct values upon corruption.

[Hof08] shows the impossibility of a realization of selective decommitment. This means that encryption needs to be non-committing to work in the setting of dynamic corruption, so it *is* possible to circumvent this problem, but at the cost of of having to updating the key regularly [CHK04], having a key-size larger than the message space [CDNO97] or require interaction with the receiver to communicate[DN00]. All of this is impractical for our purposes, as encryption in Security APIs is typically needed to not be efficient for a possibly large number of messages encrypted with the same key, without on-line communication to

the recipient. This is the scenario we have to deal with. We would like to use traditional, efficient encryption systems.

While practical considerations make realizing $\mathcal{F}^{\text{toy}}$ more difficult in this regard, they are forgiving in another: In $\mathcal{F}^{\text{toy}}$, we do not mind if an intruder learns about the content of already produced ciphertexts when corrupts the slave key. But we do want to assure that past and future ciphertexts generated using the master key stay secret - in contrast to the idea of forward secret encryption. Unfortunately, it seems that we are not able to express *non-forward secure encryption* in existing simulation-based security frameworks. In our opinion, this is a problem of *expressiveness*: we cannot formulate our weaker notion of dynamic key-corruption in, e. g., UC.

Existing work circumvents the problem in two ways. The first option is to use a game-based definition instead simulation-based security, see for example [KSW11]. There, the attacker tries to distinguish real encryptions from fake encryption and is constraint by a number of rules, to avoid what are called "trivial attacks". In this particular case, there are five kinds of request the adversary is not allowed to make. It is very difficult to say whether any of those constraints forbids more than just "trivial attacks". The attacker has to choose which key to attack before the experiment. It is not clear whether this is a restriction. Since Security APIs are supposed to be used in other protocols, composability is a big issue. It is not clear how Security APIs behave when used in several protocols at once, so a proof by reduction to the game-based definition is necessary. The second option is conditional simulatability, i. e., an implementation is proven to UC-realize $\mathcal{F}^{\text{toy}}$ or $\mathcal{F}^{\text{KW}}$ for a restricted set of environments $\mathcal{E}$, most often for all $\mathcal{E}$ that do not corrupt keys that have been used to encrypt before corruption. Theorem 16 shows a similar property. Although this approach benefits from some advantages of simulation-based security, it turns out to be insufficient for Security APIs. The restriction on the environment may hold for most protocols, still, when a party that uses a Security API becomes corrupted, it is impossible to guarantee that it corrupts the key before it uses it to encrypt. We aim at using Security APIs as building blocks in arbitrary protocols to allow for secure execution of the protocol *despite corrupted parties*, so conditional simulatability does not solve the problem, either.

We conclude that a new framework is needed that allows to use non-forward secure encryption in a composable setting with dynamic corruption by allowing to express that encrypted messages *must remain secure until a key is corrupted, but not necessarily longer*. A framework that could conserve the nice properties of simulation-based security(intuitiveness, composability) would be of great benefit. Furthermore, we suspect that this kind of "weak" dynamic corruption might turn out to be equal to static corruption.

In the full version of the paper [KKS12] we prove that a deterministic authenticated symmetric key-wrapping scheme $KW = (\mathsf{KG}, \mathsf{Wrap}, \mathsf{UnWrap})$, as defined by Kremer et al. [KSW11] can be transformed into a protocol version $\hat{\pi}^{KW}$ that UC-emulates $\mathcal{F}^{\text{KW}}$ under the condition *corrupt-before-wrap*.

B The Full Proof of Theorem 12

Theorem 12. *ST is secure with respect to any parameters* $(\Phi, \Phi^{\text{ext}}, \mathit{Room}, \overline{\mathcal{F}}, \Pi)$*, where* Π *describes a static-key hierarchy.*

Proof. In the following, we will abbreviate $\mathcal{F}^{\text{KM}}-\text{Net}_{(\Phi,\Phi^{\text{ext}},\mathit{Room},\overline{\mathcal{F}},\Pi)}$ by $\mathcal{F}^{\text{KM}}-\text{Net}$ and $\text{STN}_{ST,\Phi,\Phi^{\text{ext}},\mathit{Room},\overline{\mathcal{F}},\Pi}$ by STN. We have to show that there is a simulator *Sim*, such that for all environments $\mathcal{E}$, $\Pr[\text{EXEC}_{\text{STN},\mathcal{D},\mathcal{E}} = 1]$ - $\Pr[\text{EXEC}_{\mathcal{F}^{\text{KM}}-\text{Net},\mathit{Sim},\mathcal{E}} = 1]$ is negligible. Since the environment can access the Secure Tokens through the dummy users Φ, it does not need to corrupt them. The Security APIs themselves are assumed to be tamper-resistant. Technically, if $\mathcal{D}$ corrupts a dummy user φ_i (which is stateless), $\mathcal{E}$ can instruct it to send messages to other parties than ST_i, but other $ST_j, j \neq i$ will respond with $\perp$. Therefore, when *Sim* is instructed by the environment to send something on behalf of the corrupted user U_i in the ideal world, it will return $\perp$ if it is addressed to ST_j for $j \neq i$. If the message is addressed to ST_i, *Sim* will append the string $\texttt{by}(U_i)$ to the message. By definition of $\mathcal{F}^{\text{KM}}$, this is will be treated as if the message were coming from the dummy party for U_i, therefore we conclude that, without loss of generality, we can assume that $\mathcal{E}$ never corrupts any $U \in \mathcal{U}$. Aside from the behaviour we specified, *Sim* behaves like $\mathcal{D}$, e. g., it might forward messages to the KU-functionalities.

Before proving the claim formally, we introduce some notation: When we talk about internal variables of functionalities or protocols, we subscript the variable with the entity it belongs to whenever we need to disambiguate them. For example, $\texttt{credentials}(i)_{\mathcal{F}^{\text{KM}}}$ denotes the credentials for index i in the key-management functionality, and not the credentials inside a secure token.

Invariant As soon as $to(m_k)$ has finished processing m_k, the following holds for all sequences of messages $(m_0,\ldots,m_k)$ the environment sends or receives via the dummy users or the adversary:

- *state consistency*: For all users U_j and handles h: If $i = I(U_j, h)$ is defined on $\mathcal{F}^{\text{KM}}$, then **(a)** $\texttt{credentials}(i)_{\mathcal{F}^{\text{KM}}}$ and $\texttt{credentials}(h)_{ST_j}$ are equally distributed and **(b)** $\texttt{attr}(U_j, h)_{\mathcal{F}^{\text{KM}}} = \texttt{attr}(h)_{ST_j}$. If $I(U_j, h)$ is undefined, then no entry with h as first element exists in $\texttt{List}^{\text{KM}}_{\mathcal{F}}$. A corruption request for (U_j, h) is recorded on ST_j iff and only iff a corruption request for (U_j, h) is recorded on $\mathcal{F}^{\text{KM}}$.
- *phase consistency:* $\mathcal{F}^{\text{KM}}$ and all $ST_i \in \mathit{Room}$ are in the same phase.
- *secrecy of uncorrupted wrapping credentials:* If, $\texttt{attr}(U_j, h)_{\mathcal{F}^{\text{KM}}} = 0$, $c = \texttt{credentials}(I(U_j, h))$ and there is $[(ptr, k) \in \texttt{List}$ s.t. $k \in \mathcal{K}_{\text{uncorrupted}}]_{\mathcal{F}^{\text{KW}}}$, then, the output of ST_j and $\mathcal{F}^{\text{KM}}$ is independent of c.
- *output consistency:* The distribution of all messages received by the environment is the same in the real execution and the ideal execution
- *consistency of functionalities:* $\mathcal{F}_1 \ldots \mathcal{F}_l$ and $\mathcal{F}^{\text{KW}}$ have equally distributed states in both the real execution and the ideal execution

Initial step: For the empty sequence, i. e., prior to all messages sent by the environment, the state of all ST_i is empty, as well as the state of $\mathcal{F}^{\text{KM}}$, and they

are in the init phase. Furthermore, no message is output to the environment, and the functionalities did not receive any messages yet, therefore we conclude that the invariant is preserved.
Induction step Now assume that the invariant holds after messages $(m_0, \ldots, m_{k-1})$. The environment sends a message m_k. Case distinction:
Case: to$(m_k) \in \{F_1 \cdots F_l\}$: (Only $\mathcal{A}$ and the external dummy users in Φ^{ext} can address the KU-functionalities.) Neither state, nor phase of $\mathcal{F}^{\text{KM}}$ and any ST are affected, thus state consistency and phase hold. Both $\mathcal{F}^{\text{KM}}$ and the ST_i ignore messages coming from the functionalities, except while processing a Command $\in \mathcal{C}$. Consistency of functionalities and output consistency is preserved, because the functionalities are equal in both worlds and receive the same input. Hence, consistency of functionalities follows from the induction hypothesis.
Case: to$(m_k) = ST_i$ *and phase set-up*

- *m=(New,sid,F,a) while (F=*$\mathcal{F}^{\text{KW}}$ *and a>0) or (F*$\in \{F_1 \cdots F_l\}$ *and* $a = 0$*)* The same sequence of commands is sent to $\mathcal{F}$ in both executions. By consistency of functionalities in the previous proof step, and the fact that $\mathcal{F}$ runs the same code in both worlds, we can assure consistency of functionalities. The credentials output by $\mathcal{F}$ are saved in $\texttt{credentials}(i)_{\mathcal{F}^{\text{KM}}}$ and $\texttt{credentials}(h)_{ST_i}$, respectively. By state consistency in the previous step, and since $\mathcal{F}$ receives the same commands in both instances, the distribution of the credentials is equal. Thus, state consistency holds. Attributes are stored correctly. Output and phase consistency, as well as secrecy of uncorrupted wrapping credentials, hold trivially.
- *m=(Share,sid,*(U_i, h_1)*,*(U_j, h_2)*) and* U_i *=recipient*(m) For this case we have to look into the details of the scheduling of messages in the UC-model. When a message is sent on a direct channel, the next program called is the code that processes the message. This is why the message sent via $\mathcal{F}^{\text{SIMT}}$ to ST_j is processed before the environment can send any further messages. By inspection of the messages sent and the fact that $\mathcal{F}^{\text{SIMT}}$ guarantees authenticity, we see that credentials, attributes and types are correctly transferred between ST_i and ST_j, thus state consistency holds.
  If $I(U_j, h_2)^{\text{KM}}_{\mathcal{F}}$ is undefined or there is no List entry for h in ST_j, $\{0, 1\}$ is output, otherwise (Share,*sid*). By state consistency, those two events coincide, i. e., in both networks the same is output, thus output consistency holds. The phase is not changed, so the invariant holds in this case.

Case: to$(m_k) = ST_i$ *and phase "run"* Once in phase "run", no phase change takes place anymore in every $\mathcal{F}^{\text{KM}}$ and ST_i, therefore phase consistency holds for any m.

- *m=(New,sid,F,h,a)* The handling of this message is defined just as in the set-up phase, and since the invariant is independent of the phase we are in (except for phase consistency, of course, but we just dealt with this) , the argument is literally the same as for the same message in the set-up phase.
- *m=(AttributeChange,sid,h,a_new)* The condition in ST_i implements the policy relation, therefore the invariant transfers easily.

- *m=(Wrap,sid,h_1, h_2,id)* State consistency holds trivially by the fact that neither credentials, nor attributes are touched. Therefore, the wrapping functionality $\mathcal{F}^{\mathrm{KW}}$ receives the same messages in both worlds and is, by consistency of functionalities in the previous proof step, in the same state in both worlds at the end of its processing of the message. Thus consistency of functionalities holds, and by the fact that the response is forwarded in the same manner, output consistency holds as well. Since $\mathcal{F}^{\mathrm{KW}}$ only outputs the message to be wrapped iff the wrapping keys is corrupted, we can assure secrecy of uncorrupted wrapping credentials. If the handles h_1 or h_2 are not defined, by state consistency, both $\mathcal{F}^{\mathrm{KM}}$ and ST_i output $\perp$.
- *m=(Unwrap,sid,h,c, a, F, id)*
  - $I(U_j, h)_{\mathcal{F}^{\mathrm{KM}}}$ *not defined, or policy(*$\mathcal{F}^{\mathrm{KW}}$*,Unwrap,* $[\mathtt{attr}(I(U,h))]_{\mathcal{F}^{\mathrm{KM}}}$*, a) = false* Due to state consistency, in both worlds $\perp$ is output and nothing else, thus the invariant is preserved.
  - $\mathcal{F}^{\mathrm{KW}}$ *returns* $\perp$ *to the query (Unwrap,sid,* $\mathtt{credentials}(I(U,h)$*, <a , type, id>,w).* by state consistency, the query equals the query (*Unwrap*,*sid*, $\mathtt{credentials}(h)$, <a,type,id>,w) emitted by ST_i, hence, by consistency of functionalities, ST_i receives $\perp$, too. Therefore, both respond with $\perp$.
  - *otherwise:* In $\mathcal{F}^{\mathrm{KM}}$ a new handle is created, and $I(U,h)$ updated to either point to a newly generated key-index, or to one already in place with the same credentials and attributes. In both cases, ST_i creates a new handle and stores it along credential, type and attribute in the list. Hence, state consistency holds. Secrecy of uncorrupted wrapping credentials and output consistency follow from the fact that no output is produced except on the secret channel to $\mathcal{F}^{\mathrm{KW}}$. Consistency of functionalities follows from the fact that the same unwrap command is sent to $\mathcal{F}^{\mathrm{KW}}$ in both cases, and that the credentials are equal by state consistency from the previous proof step.
- *m=(Cmd,sid,h,m)* No internal state is changed, the message is directly forwarded, so phase consistency and state consistency hold. Because $\mathcal{F}^{\mathrm{KW}} \notin \{\mathcal{F}_1, \ldots, \mathcal{F}_l\}$, the output is independent of each wrapping credential, so secrecy of uncorrupted wrapping credentials is preserved. State consistency guarantees that the same message to $\mathcal{F}$ is computed. Since consistency of functionalities in the previous proof step can be assumed, and key-manageable functionalities are caller-independent (see Definition 4) , we conclude consistency of functionalities for this step. Output consistency follows, because both ST_j and $\mathcal{F}^{\mathrm{KM}}$ only forward the output of $\mathcal{F}$.
- *m=(*Corrupt*,sid,*(U_j, h)*)* By state consistency, the response is distributed equally in real and ideal execution, hence output consistency holds, and because the output is either $\perp$ or depends only on the value of a corrupted credential, secrecy of corrupted credentials is guaranteed as well. Neither changes its internal state, except for the register of corruption request: the corruption request for (U_j, h) is recorded on ST_j as well as on $\mathcal{F}^{\mathrm{KM}}$ if $I(U_j, h)$ and $(h, \mathtt{credentials}(h))$ are defined, which coincides by state consistency of the previous step. Therefore, state consistency holds for the next step.

The message is addressed to the adversary only, so we have consistency of functionalities.

- m=(CorruptionStatus,*sid*,(*U*,*h*)) By state consistency, the response is the same in the real and the ideal execution. No internal state is changed, therefore, the invariant holds for this step.
- *m is none of the above* $\mathcal{F}^{\mathrm{KM}}$ and ST_i output $\perp$.

C $\mathcal{F}^{\mathrm{KW}}$ Can Be Implemented Using Deterministic Authenticated Encryption with Key-Dependant Security

In the following we describe a Key-wrapping functionality and show that it can be implemented using a deterministic authenticated encryption scheme as defined in [RS06].

Definition 17. *Given* KW=(KG, Wrap, UnWrap), *we define the protocol version* $\hat{\pi}^{KW}$ *as follows:*

params: KW = (KG, *Wrap*, *Unwrap*), *distribution of pointers* $\mathcal{P}$
structure: *List of* $\{0,1\}^{\eta} \times$ keys
phase init:
rcv *(Algorithms, sid, wrap, unwrap)* ***from*** $\mathcal{A}$
    enter phase *run*
phase *run:*
rcv *(Keygen, sid)* ***from*** *P*
    save $(ptr \leftarrow \mathcal{P}, k \leftarrow \mathrm{KG}(1^{\eta}))$ ***in*** *List*
    snd *(KeyPointer, sid, ptr)* ***to*** *P*
rcv *(Wrap,sid, ptr,a, m)* ***from*** *P*
    search *k* ***such that*** *(ptr,k)* ***in*** *List*
    if $m \in \mathcal{P}$
        search *(m,k')* ***in*** *List*
        snd $(Wrap^{\bullet}, Wrap_k^a(k'))$ ***to*** *P*
    else snd $(Wrap^{\bullet}, Wrap_k^a(m))$ ***to*** *P*
rcv *(Unwrap,sid, ptr,a,c)* ***from*** *P*
    search *k* ***such that*** *(ptr,k)* ***in*** *List*
    snd $(Unwrap^{\bullet}, Unwrap_k^a(c))$ ***to*** *P*
rcv *(Corrupt,sid,ptr)* ***from*** $\mathcal{A}$
    search *k* ***such that*** *(ptr,k)* ***in*** *List*
    save *corruption request for ptr*
    snd (Corrupt$^{\bullet}$,*sid*,*k*) ***to*** $\mathcal{A}$
rcv *(CorruptionStatus,sid,ptr* $\in \mathcal{P}$*)* ***from*** *P*
    if *corruption request for ptr was saved*
        snd (CorruptionStatus,*sid*,*Corrupted*) ***to*** *U*
    else snd $\perp$ ***to*** *U*

Listing 1.10. $\hat{\pi}^{KW}$ given KG,Wrap,UnWrap

We repeat the definition of from [KSW11] here. It is based on the notion of deterministic, authenticated encryption from [RS06], but it additionally supports key-dependant messages. We changed the definition, so it allows to wrap the same key with the same wrapping key but under different attributes, just like in the DAE definition from [RS06].

Definition 18 (Multi-user setting for key wrapping). *We define experiments* $\mathbf{Exp}^{\mathsf{wrap,real}}_{\mathcal{A},\mathsf{KW}}(\eta)$ *and* $\mathbf{Exp}^{\mathsf{wrap,fake}}_{\mathcal{A},\mathsf{KW}}(\eta)$*. In both experiments the adversary can access a number of keys* $k_1, k_2, \ldots, k_n \ldots$ *(which he can ask to be created via a query* NEW*). In his other queries, the adversary refers to these keys via symbols* $K_1, K_2, \ldots, K_n$ *(where the implicit mapping should be obvious). By abusing notation we often use* K_i *as a placeholder for* k_i *so, for example,* $\mathsf{Wrap}^a_{K_i}(K_j)$ *means* $\mathsf{Wrap}^a_{k_i}(k_j)$*. We now explain the queries that the adversary is allowed to make, and how they are answered in the two experiments.*

- $\mathsf{NEW}(K_i)$*: a new key* k_i *is generated via* $k_i \leftarrow \mathsf{KG}(\eta)$
- $\mathsf{ENC}(K_i, a, m)$ *where* $m \in \mathcal{K} \cup \{K_i \mid i \in \mathbb{N}\}$ *and* $h \in \mathcal{H}$*. The experiment returns* $\mathsf{Wrap}^a_{k_i}(m)$*.*
- $\mathsf{TENC}(K_i, a, m)$ *where* $m \in \mathcal{K} \cup \{K_i \mid i \in \mathbb{N}\}$ *and* $a \in \mathcal{H}$*. The real experiment returns* $\mathsf{Wrap}^a_{k_i}(m)$*, whereas the fake experiment returns* $\$^{|\mathsf{Wrap}^a_{k_i}(m)|}$
- $\mathsf{DEC}(K_i, a, c)$*: the real experiment returns* $\mathsf{UnWrap}^a_{k_i}(c)$*, the fake experiment returns* $\perp$*.*
- $\mathsf{CORR}(K_i)$*: the experiment returns* k_i

Correctness of the wrapping scheme requires that for any $k_1, k_2 \in \mathcal{K}$ and any $a \in \mathcal{H}$, if $c \leftarrow \mathit{Wrap}^a_{k_1}(k_2)$ then $\mathit{Unwrap}^a_{k_1}(c) = k_1$.

Consider the directed graph whose nodes are the symbolic keys K_i and in which there is an edge from K_i to K_j if the adversary issues a query $\mathsf{ENC}(K_i, a, K_j)$. We say that a key K_i is corrupt if either the adversary corrupted the key from the start, or if the key is reachable in the above graph from a corrupt key. If a handle, respectively pointer, points to a corrupted key, we call the pointer corrupted as well.

We make the following assumptions on the behaviour of the adversary.

- For all i the query $\mathsf{NEW}(K_i)$ is issued at most once.
- All the queries issued by the adversary contain keys that have already been generated by the experiment.
- The adversary never makes a test query $\mathsf{TENC}(K_i, a, K_j)$ if K_i is corrupted at the end of the experiment.
- If A issues test query $\mathsf{TENC}(K_i, a, m)$ then A does not issue $\mathsf{TENC}(K_j, a', m')$ or $\mathsf{ENC}(K_j, a', m')$ with $(K_i, m) = (K'_j, m')$
- The adversary never queries $\mathsf{DEC}(K_i, a, c)$ if c was the result of a query $\mathsf{TENC}(K_i, a, m)$ or of a query $\mathsf{ENC}(K_i, a, m)$ or K_i is corrupted.

At the end of the execution the adversary has to output a bit b which is also the result of the experiment. The advantage of adversary A in breaking the

key-wrapping scheme KW is defined by:

$$Adv^{\mathsf{wrap}}_{\mathsf{KW},A}(\eta) = \Big|\Pr\left[b \leftarrow \mathbf{Exp}^{\mathsf{wrap,real}}_{\mathsf{KW},A}(\eta) \ : b = 1\right] - \Pr\left[b \leftarrow \mathbf{Exp}^{\mathsf{wrap,fake}}_{\mathsf{KW},A}(\eta) \ : b = 1\right]\Big|$$

and KW is secure if the advantage of any probabilistic polynomial time algorithm is negligible.

Theorem 16. *If a symmetric encryption scheme $KW = (\mathsf{KG}, \mathsf{Wrap}, \mathsf{UnWrap})$ is secure with respect to Definition 18, then for the protocol version $\hat{\pi}^{KW}$ (see Definition 17), $\hat{\pi}^{KW}$ UC-emulates $\mathcal{F}^{\mathrm{KW}}$ under the condition corrupt-before-wrap.*

Proof. Assume for contradiction an environment $\mathcal{E}$ such that $\Pr[\mathrm{EXEC}_{\mathcal{E},\mathcal{D},\hat{\pi}^{KW}} = 1|$ *corrupt-before-wrap*$]$ - $\Pr[\mathrm{IDEAL}_{\mathcal{E},Sim,\mathcal{F}^{\mathrm{KW}}} = 1|$ *corrupt-before-wrap*$]$ is non-negligible for all simulators. We fix the following simulator:

phase init:
  snd (Algorithms, sid, *Wrap*, *Unwrap*) **to** FKW **and enter phase** run
phase run:
  rcv (Generate,*sid*) **from** $\mathcal{F}^{\mathrm{KW}}$
    snd (Generate,*sid*,$k \leftarrow \mathrm{KG}(1^\eta)$) **to** $\mathcal{F}^{\mathrm{KW}}$
  rcv (*Wrap*,*sid*,*ptr*,*k*,a,m) **from** $\mathcal{F}^{\mathrm{KW}}$
    snd (*Wrap*,*sid*,$Wrap^a_k(m)$) **to** $\mathcal{F}^{\mathrm{KW}}$
  otherwise: act like $\mathcal{D}$

We now generate an adversary $\mathcal{B}$ against Definition 18. Internally, it simulates $\mathcal{E}$, acting for the dummy-adversary, respectively the adversary-simulator, as well as for $\hat{\pi}^{KW}$, respectively $\mathcal{F}^{\mathrm{KW}}$.

- If $\mathcal{E}$ sends (Generate, *sid*) to $\mathcal{F}^{\mathrm{KW}}$, $\mathcal{B}$ echoes the same message to $\mathcal{E}$, pretending to be *Sim*. It waits for the environment's response of form (Generate,*sid*,k') and ignores it. As we are not yet sure whether this key will become corrupted, we do not emit a NEW query yet. Instead, $\mathcal{B}$ draws a pointer $ptr \leftarrow \mathcal{P}$ and saves it in the list of uninitialized pointers $\mathcal{P}_{new}$. $\mathcal{B}$ sends (Generate$^\bullet$,*sid*,*ptr*) to $\mathcal{E}$. If any of the following commands takes a key-pointer *ptr*, output ($\perp$) in case *ptr* was not generated in this step.
- If $\mathcal{E}$ sends (CorruptionStatus,*sid*,*ptr*) to $\mathcal{F}^{\mathrm{KW}}$ for a pointer for which it sent a (Corrupt,*sid*,*ptr*) query before, the output is (CorruptionStatus$^\bullet$, *sid*,Corrupted), otherwise, the output is $\perp$.
- If $\mathcal{E}$ sends (Corrupt,*sid*,*ptr*) to $\mathcal{F}^{\mathrm{KW}}$ for an uninitialized pointer *ptr*, $\mathcal{B}$ emits a query $\mathsf{NEW}(K_{ptr}, 1)$. Save the response in k_{ptr}. Record *ptr* as initialized and corrupted. Record the corruption query for this pointer, to answer later (CorruptionStatus) queries correctly. If any of the following commands is called for a key-pointer *ptr* that is uninitialized, then emit a query $\mathsf{NEW}(K_{ptr}, 0)$ and record *ptr* as initialized and uncorrupted before further processing it.

– If $\mathcal{E}$ sends ($Wrap$,sid,ptr,a,m) to $\mathcal{F}^{\mathrm{KW}}$ for an uncorrupted pointer ptr, $\mathcal{B}$ checks if the query TENC (K_{ptr},a, m) was emitted before. If this is true, return the response r to this query saved previously to $\mathcal{E}$, in form of the message ($Wrap^{\bullet}$,r). If not, the query is emitted, and the response r sent to $\mathcal{E}$ in the same form, whilst being saved as a response to the query TENC (K_{ptr},a, m). If ptr is corrupted, compute $c := Wrap_k^a(m)$ for the k associated to ptr. Return ($Wrap^{\bullet}$, r) to $\mathcal{E}$, where r is the response again. In case $m \in \mathcal{P}$, if m is uninitialized, $\mathcal{B}$ emits a query $\mathsf{NEW}(K_m, 1)$, saves the response in k_m and records m as initialized and corrupted.
– If $\mathcal{E}$ sends ($Unwrap$,sid,ptr,a,c) to $\mathcal{F}^{\mathrm{KW}}$ for a ptr that is marked uncorrupted, then $\mathcal{B}$ responds by sending ($Unwrap^{\bullet}$, sid, m) for a message m previously recorded in a query TENC (ptr,a,m) with response c. If no such query exists, call DEC (K_{ptr},a,c), and use the response in place of m. In case ptr is corrupted, send ($Unwrap^{\bullet}$, sid, $Unwrap_{k_{ptr}}^a(c)$) to $\mathcal{E}$.

First we show that the adversary $\mathcal{B}$ is valid, in the sense that it does not violate any of the five assumptions on his behaviour from Definition 18, then we show that we provide $\mathcal{E}$ with suitable views for the case of the fake, as well as the real game. First, the probability that $(\perp, Collision)$ is output is negligible. If this is never output, we can guarantee that all $\mathsf{NEW}(K_i)$ queries have distinct i. Second, before issuing a query, it is checked if the key-pointer has previously been initialized using a NEW query. Third, $\mathsf{TENC}(K_{ptr}, a, m)$ is only queried if K_{ptr} is uncorrupted. Fourth, ENC is never queried, and $\mathcal{B}$ remembers the response to a previous TENC query, so it does not issue it twice. Fifth, assume ptr to be uncorrupted. Then, a ciphertext is only part of in a DEC query, if it did not result from a TENC query to the same functionality before. If ptr is corrupted, no DEC query is sent. Note that we never emit a ENC-query. Since, by the fourth condition, it cannot be used on a key that is later used for a test query, it does not provide for more than an oracle $\mathsf{Wrap}_k^{\cdot}(\cdot)$ for a randomly drawn $k \leftarrow \mathrm{KG}(1^\eta)$

We now show that, for the real experiment, the view of $\mathcal{E}$ is the same as in the real execution. (Generate,sid) generates a random handle $ptr \in \mathcal{P}$ and no further output. A key k_{ptr} is drawn using KG as soon as ptr is used in another query for the first time. In the real experiment, ($Wrap$,sid,ptr,a,m) outputs $Wrap_{k_{ptr}}^a(m)$, performing the same substitution of ptr by k_{ptr} that the experiment performs by "abuse of notation". Similarly, ($Unwrap$, sid, ptr, a, c) outputs $Unwrap_{k_{ptr}}^a(c)$, except in case ptr points to an uncorrupted key k and c resulted from a previous query $\mathsf{TENC}(K_{ptr}, a, m)$. $\mathcal{B}$ responds with the message m that was queried. In this case, by the correctness property we have that $m = Unwrap_k^a(c)$. If (Corrupt,sid,ptr) is called, $\mathcal{E}$ receives the key k_{ptr} that will be used for all further wrappings and unwrappings for the pointer ptr. If (CorruptionStatus,sid,ptr) is called, $\mathcal{B}$ outputs, independent of the experiment, and like both $\mathcal{F}^{\mathrm{KW}}$ and $\hat{\pi}^{KW}$, whether the pointer is initialized and a corruption query for this pointer was emitted before. Therefore, $\mathcal{E}$ receives the same output that $\hat{\pi}^{KW}$ is defined to produce.

Now we show that, in the fake experiment, the view of $\mathcal{E}$ is the same as in the ideal execution. (Generate,sid) behaves just as before and outputs a

randomly drawn pointer. With the same argument as before, (CorruptionStatus) is simulated correctly. (*Wrap*,*sid*,*ptr*,*a*,*m*) outputs $Wrap^a_{k_{ptr}}(L^a(k_{ptr}, m))$ (m might be a key or a message) if *ptr* is uncorrupted, otherwise $\mathcal{B}$ computes $Wrap^a_{k_{ptr}}(m)$ without emitting a query, providing the same outputs as $\mathcal{F}^{\mathrm{KW}}$ for corrupted, as well as uncorrupted keys. Again, pointers are substituted by keys in this step. (*Unwrap*, *sid*, *ptr*, *a*, *c*) is a more complicated case: If k_{ptr} is an uncorrupted key and r was the result of a previous (*Wrap*,*ptr*,*a*,*m*) query, just as $\mathcal{F}^{\mathrm{KW}}$, the simulation outputs m. If c is uncorrupted, but such a query has not been made before, $\mathsf{DEC}(K_{ptr}, a, c)$ outputs $\perp$, and (*Unwrap*$^\bullet$,*sid*,$\perp$) is output, exactly as $\mathcal{F}^{\mathrm{KW}}$ is specified to do. If c is corrupted, the simulation, as well as $\mathcal{F}^{\mathrm{KW}}$ compute (*Unwrap*$^\bullet$, *sid*, $Unwrap^a_{k_{ptr}}(c)$).

Since we have established that $\mathcal{B}$ is a valid adversary and, as argued before, $\mathcal{E}$ is provided with correct views, we have that $\Pr[b \leftarrow \mathbf{Exp}^{\mathsf{wrap,real}}_{\mathsf{KW},\mathcal{B}}(\eta) \ : b = 1]$ equals $\Pr[\mathrm{EXEC}_{\mathcal{E},\mathcal{D},\hat{\pi}^{KW}} = 1|$ *corrupt-before-wrap*], as well as $\Pr[b \leftarrow \mathbf{Exp}^{\mathsf{wrap,fake}}_{\mathsf{KW},\mathcal{B}}(\eta) \ : b = 1]$ equals $\Pr[\mathrm{IDEAL}_{\mathcal{E},Sim,\mathcal{F}^{\mathrm{KW}}} = 1|$ *corrupt-before-wrap*]. Thus, we can conclude that $Adv^{\mathsf{wrap}}_{\mathsf{KW},A}(\eta)$ is non-negligible, contradicting the assumption.